Chapter 5

Advanced Optimization Solutions

"The journey of a thousand miles begins with one step." –Lao Tzu

5.1 Introduction

The previous chapter introduced several basic algorithms for gradient descent. However, these algorithms do not always work well because of the following reasons:

- *Flat regions and local optima:* The objective functions of machine learning algorithms might have local optima and flat regions in the loss surface. As a result, the learning process might be too slow or arrive at a poor solution.
- *Differential curvature:* The directions of gradient descent are only *instantaneous* directions of best movement, which usually change over steps of finite length. Therefore, a steepest direction of descent no longer remains the steepest direction, after one makes a finite step in that direction. If the step is too large, the different components of the gradient might flip signs, and the objective function might worsen. A direction is said to show high *curvature*, if the gradient changes rapidly in that direction. Clearly, directions of high curvature cause uncertainty in the outcomes of gradient descent.
- *Non-differentiable objective functions:* Some objective functions are non-differentiable, which causes problems for gradient descent. If differentiability is violated at a relatively small number of points and the loss function is informative for the large part, one can use gradient descent with minor modifications. More challenging cases arise when the objective functions have steep cliffs or flat surfaces in large regions of the space, and the gradients are not informative at all.

The simplest approach to address both flat regions and differential curvature is to adjust the gradients in some way to account for poor convergence. These methods *implicitly* use the curvature to adjust the gradients of the objective function with respect to different parameters. Examples of such techniques include the pairing of vanilla gradient-descent methods with computational algorithms like the *momentum method*, *RMSProp*, or *Adam*.

C. C. Aggarwal, *Linear Algebra and Optimization for Machine Learning*,
https://doi.org/10.1007/978-3-030-40344-7_5

Another class of methods uses second-order derivatives to *explicitly* measure the curvature; after all, a second derivative is the rate of *change* in gradient, which is a direct measure of the unpredictability of using a constant gradient direction over a finite step. The second-derivative matrix, also referred to as the *Hessian*, contains a wealth of information about directions along which the greatest curvature occurs. Therefore, the Hessian is used by many second-order techniques like the *Newton method* in order to adjust the directions of movement by using a trade-off between the steepness of the descent and the curvature along a direction.

Finally, we discuss the problem of non-differentiable objective functions. Consider the L_1-loss function, which is non-differentiable at some points in the parameter space:

$$f(x_1, x_2) = |x_1| + |x_2|$$

The point $(x_1, x_2) = (0, 0)$ is a non-differentiable point of the optimization. This type of setting can be addressed easily by having special rules for the small number of non-differentiable points in the space. However, in some cases, non-informative loss surfaces contain only flat regions and vertical cliffs. For example, trying to directly optimize a ranking-based objective function will cause non-differentiability in large regions of the space. Consider the following objective function containing training points $\overline{X}_1 \ldots \overline{X}_n$, of which a subset S belong to a *positive class* (e.g., fraud instances versus normal instances):

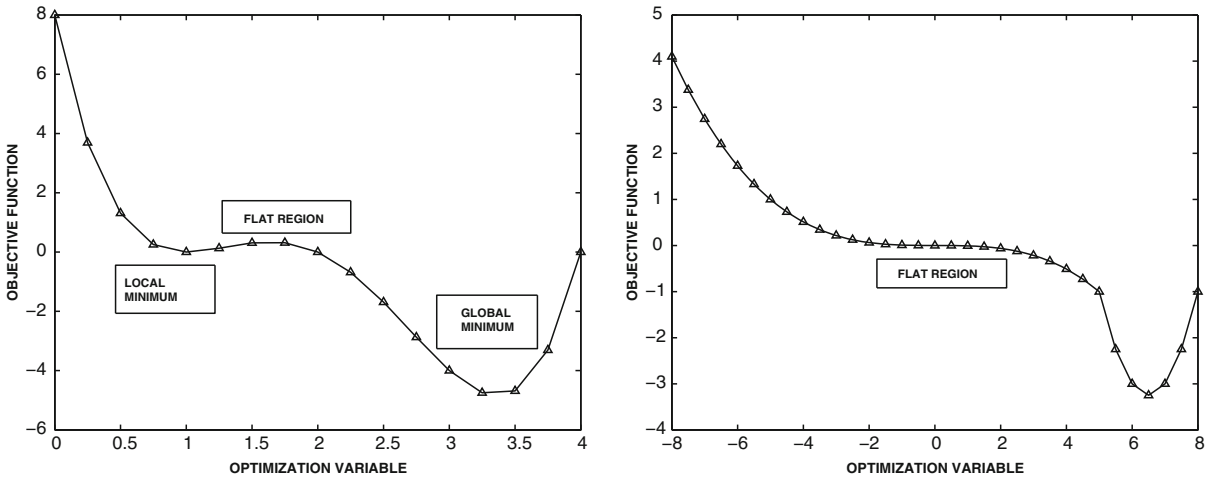
$$J(\overline{W}) = \sum_{i \in S} \text{Rank}(\overline{W} \cdot \overline{X}_i)$$

Here, the function "Rank" simply computes a value from 1 through n, based on sorting the values of $\overline{W} \cdot \overline{X}_i$ over the n training points and returning the rank of each $\overline{X}_i$. Minimizing the function $J(\overline{W})$ tries to set $\overline{W}$ to ensure that positive examples are always ranked before negative examples. This kind of objective function will contain only flat surfaces and vertical cliffs with respect to $\overline{W}$, because the ranks can suddenly change at specific values of the parameter vector $\overline{W}$. In most regions, the ranks will not change on perturbing $\overline{W}$ slightly, and therefore $J(\overline{W})$ will have a zero gradient in most regions. This type of setting can cause serious problems for gradient descent because the gradients are not informative at all. In such cases, more complex methods like the proximal gradient method need to be used. This chapter will discuss several such options.

This chapter is organized as follows. The next section will discuss the challenges associated with optimization of differentiable functions. Methods that modify the first-order derivative of the loss function to account for curvature are discussed in Section 5.3. The Newton method is introduced in Section 5.4. Applications of the Newton method to machine learning are discussed in Section 5.5. The challenges associated with the Newton method are discussed in Section 5.6. Computationally efficient approximations of the Newton method are discussed in Section 5.7. The optimization of non-differentiable functions is discussed in Section 5.8. A summary is given in Section 5.9.

5.2 Challenges in Gradient-Based Optimization

In this section, we will discuss the two main problems associated with gradient-based optimization. The first problem has to do with flat regions and local optima, whereas the second problem has to do with the different levels of curvature in different directions. Understanding these problems is one of the keys in designing good solutions for them. Therefore, this section will discuss these issues in detail.



(a) Local optima with flat regions (b) Only global optimum with flat region

Figure 5.1: Illustrations of local optima and flat regions

5.2.1 Local Optima and Flat Regions

The previous chapter discussed several optimization models that correspond to convex functions, which have a single global optimum and no local optima. However, more complex machine learning settings like neural networks are typically not convex, and they might have multiple local optima. Such local optima create challenges for gradient descent.

Consider the following 1-dimensional function:

$$F(x) = (x-1)^2[(x-3)^2 - 1]$$

Computing the derivative and setting it to zero yields the following condition:

$$F'(x) = 2(x-1)[(x-1)(x-3) + (x-3)^2 - 1] = 0$$

The solutions to this equation are $x = 1, \frac{5}{2} - \frac{\sqrt{3}}{2} = 1.634, \frac{5}{2} + \frac{\sqrt{3}}{2} = 3.366$. From the second-derivative conditions, it is possible to show that the first and third roots are minima with $F''(x) > 0$, whereas the second root is a maximum with $F''(x) < 0$. When the function $F(x)$ is evaluated at these points, we obtain $F(1) = 0$, $F(1.634) = 0.348$, and $F(3.366) = -4.848$. The plot of this function is shown in Figure 5.1(a). It is evident that the first of the optima is a local minimum, whereas the second is a local maximum. The last point $x = 3.366$ is the *global* minimum we are looking for.

In this case, we were able to solve for both the potential minima by using the optimality condition, and then plug in these values to determine which of them is the global minimum. But what happens when we try to use gradient descent? The problem is that if we start the gradient descent from any point less than 1.634, one will arrive at a local minimum. Furthermore, one might never arrive at a global minimum (if we always choose the wrong starting point in multiple runs), and there would be no way of knowing that a better minimum exists. This problem becomes even more severe when there are multiple dimensions, and the number of local minima proliferate. We point the reader to Problem 4.2.4 of the previous chapter as an example of how local minima proliferate exponentially fast with increasing dimensionality. It is relatively easy to show that if we have d univariate functions (in different variables $x_1 \ldots x_d$), so that the ith function has k_i local/global minima, then

the d-dimensional function created by the sum of these functions has $\prod_{i=1}^{d} k_i$ local/global minima. For example, a 10-dimensional function, which is a sum of 10 instances of the function represented in Equation 5.2.1 (over different variables) would have $2^{10} = 1024$ minima obtained by setting each of the 10 dimensions to any one of the values from $\{1, 3.366\}$. Clearly, if one does not know the number and location of the local minima, it is hard to be confident about the optimality of the point to which gradient descent converges.

Another problem is the presence of flat regions in the objective function. For example, the objective function in Figure 5.1(a) has a flat region between a local minimum and a local maximum. This type of situation is quite common and is possible even in objective functions where there are no local optima. Consider the following objective function:

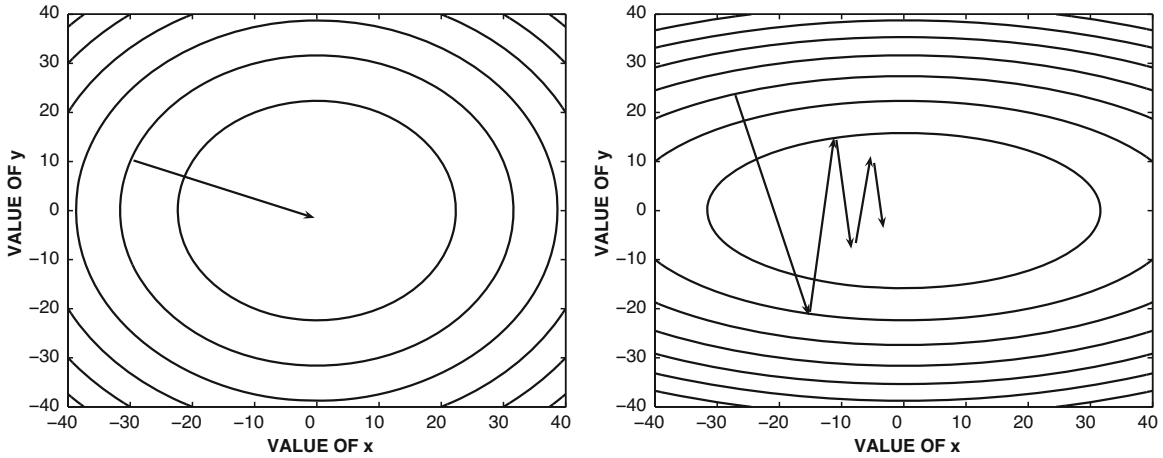
$$F(x) = \begin{cases} -(x/5)^3 & \text{if } x \leq 5 \\ x^2 - 13x + 39 & \text{if } x > 5 \end{cases}$$

The objective function is shown in Figure 5.1(b). This objective function has a flat region in the range $[-1, +1]$, where the absolute value of the gradient is less than 0.1. On the other hand, the gradient increases rapidly for values of $x > 5$. Why are flat regions problematic? The main issue is that the speed of descent depends on the magnitude of the gradient (if the learning rate is fixed). In such cases, the optimization procedure will take a long time to cross flat regions of the space. This will make the optimization process excruciatingly slow. As we will see later, techniques like momentum methods use analogies from physics in order to inherit the rate of descent from previous steps as a type of momentum. The basic idea is that if you roll a marble down a hill, it gathers speed as it rolls down, and it is often able to navigate local potholes and flat regions better because of its momentum. We will discuss this principle in more detail in Section 5.3.1.

5.2.2 Differential Curvature

In multidimensional settings, the components of the gradients may have very different magnitudes, which causes problems for gradient-descent methods. For example, neural networks often have large differences in the magnitudes of the partial derivatives with respect to parameters of different layers; this phenomenon is popularly referred to as the *vanishing and exploding gradient problem*. Minor manifestations of this problem occur even in simple cases like convex and quadratic objective functions. Therefore, we will start by studying these simple cases, because they provide excellent insight into the source of the problem and possible solutions.

Consider the simplest possible case of a convex, quadratic objective function with a bowl-like shape and a single global minimum. Two such bivariate loss functions are illustrated in Figure 5.2. In this figure, the contour plots of the loss function are shown, in which each line corresponds to points in the XY-plane where the loss function has the same value. The direction of steepest descent is always perpendicular to this line. The first loss function is of the form $L = x^2 + y^2$, which takes the shape of a perfectly circular bowl, if one were to view the height as the objective function value. This loss function treats x and y in a symmetric way. The second loss function is of the form $L = x^2 + 4y^2$, which is an elliptical bowl. Note that this loss function is more sensitive to changes in the value of y as compared to changes in the value of x, although the specific sensitivity depends on the position of the data point. In other words, the *second-order* derivatives $\frac{\partial^2 L}{\partial x^2}$ and $\frac{\partial^2 L}{\partial y^2}$ are different in the case of the loss $L = x^2 + 4y^2$. A high second-order derivative is also referred to as high curvature, because it affects how quickly the gradient changes. This is important from the perspective



(a) Loss function is circular bowl $L = x^2 + y^2$

(b) Loss function is elliptical bowl $L = x^2 + 4y^2$

Figure 5.2: The effect of the shape of the loss function on steepest-gradient descent

of gradient descent because it tells us that some directions have more consistent gradients that do not change rapidly. Consistent gradients are more desirable from the perspective of making gradient-descent steps of larger sizes.

In the case of the circular bowl of Figure 5.2(a), the gradient points directly at the optimum solution, and one can reach the optimum in a single step, as long as the correct step-size is used. This is not quite the case in the loss function of Figure 5.2(b), in which the gradients are often more significant in the y-direction as compared to the x-direction. Furthermore, the gradient never points to the optimal solution, as a result of which many course corrections are needed over the descent. A salient observation is that the steps along the y-direction are large, but subsequent steps undo the effect of previous steps. On the other hand, the progress along the x-direction is consistent but tiny. In other words, the long-term progress along each direction is very limited; therefore, it is possible to get into situations where very little progress is made even after training for a long time.

The above example represents a very simple quadratic, convex, and additively separable function, which represents a straightforward scenario compared to any real-world setting in machine learning. In fact, *with very few exceptions, the path of steepest descent in most objective functions is only an instantaneous direction of best movement, and is not the correct direction of descent in the longer term.* In other words, small steps with "course corrections" are always needed; the only way to reach the optimum with steepest-descent updates is by using an *extremely large number of tiny updates and course corrections*, which is obviously very inefficient. At first glance, this might seem almost ominous, but it turns out that there are numerous solutions of varying complexity to address these issues. The simplest example is feature normalization.

5.2.2.1 Revisiting Feature Normalization

As discussed in Chapter 4, it is common to standardize features before applying gradient descent. An important reason for scaling the features is to ensure better performance of gradient descent. In order to understand this point, we will use an example. Consider a (hypothetical) data set containing information about the classical guns-butter trade-off in

Table 5.1: A hypothetical data set of guns, butter, and happiness

Guns (number per capita)	Butter (ounces per capita)	Happiness (index)
0.1	25	7
0.8	10	1
0.4	10	4

the expenditure of various nations, together with the happiness index. The goal is to predict the happiness index y of the nation as a function of the guns per capita x_1 and the ounces per capita of butter x_2. An example of a toy data set of three points is shown in Table 5.1. A linear regression model uses the coefficient w_1 for guns and the coefficient w_2 for butter in order to predict the happiness index from guns and butter:

$$y = w_1 x_1 + w_2 x_2$$

Then, one can model the least-squares objective function as follows:

$$\begin{aligned} J &= (0.1w_1 + 25w_2 - 7)^2 + (0.8w_1 + 10w_2 - 1)^2 + (0.4w_1 + 10w_2 - 4)^2 \\ &= 0.81w_1^2 + 825w_2^2 + 29w_1 w_2 - 6.2w_1 - 450w_2 + 66 \end{aligned}$$

Note that this objective function is far more sensitive to w_2 as compared to w_1. This is caused by the fact that the butter feature has a much larger variance than the gun feature, which shows up in the coefficients of the objective function. As a result, the gradient will often bounce along the w_2 direction, while making tiny progress along the w_1 direction. However, if we standardize each column in Table 5.1 to zero mean and unit variance, the coefficients of w_1^2 and w_2^2 will become much more similar. As a result, the bouncing behavior of gradient descent is reduced. In this particular case, the interaction terms of the form $w_1 w_2$ will cause the ellipse to be oriented at an angle to the original axes. This causes additional challenges in terms of bouncing of gradient descent along directions that are not parallel to the original axes. Such interaction terms can be addressed by a procedure called *whitening*, and it is an application of the method of principal component analysis (cf. Section 7.4.6 of Chapter 7).

5.2.3 Examples of Difficult Topologies: Cliffs and Valleys

It is helpful to examine a number of specific manifestations of high-curvature topologies in loss surfaces. Two examples of high-curvature surfaces are cliffs and valleys. An example of a cliff is shown in Figure 5.3. In this case, there is a gently sloping surface that rapidly changes into a cliff. However, if one computed only the first-order partial derivative with respect to the variable x shown in the figure, one would only see a gentle slope. As a result, a modest learning rate might cause very slow progress in gently sloping regions, whereas the same learning rate can suddenly cause overshooting to a point far from the optimal solution in steep regions. This problem is caused by the nature of the curvature (i.e., changing gradient), where the first-order gradient does not contain the information needed to control the size of the update. As we will see later, several computational solutions directly or indirectly make use of second-order derivatives in order to account for the curvature. Cliffs are not desirable because they manifest a certain level of instability in the loss function. This implies that a small change in some of the weights can suddenly change the local topology so drastically that continuous optimization algorithms (like gradient descent) have a hard time.

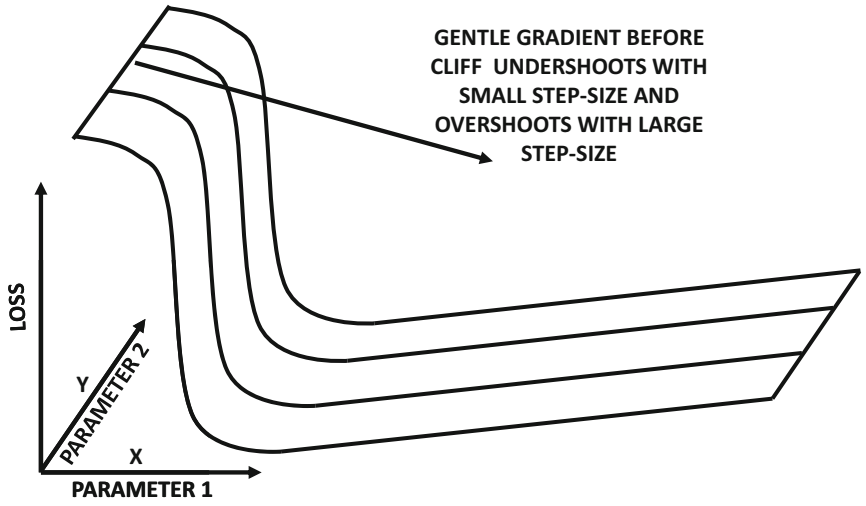


Figure 5.3: An example of a cliff in the loss surface

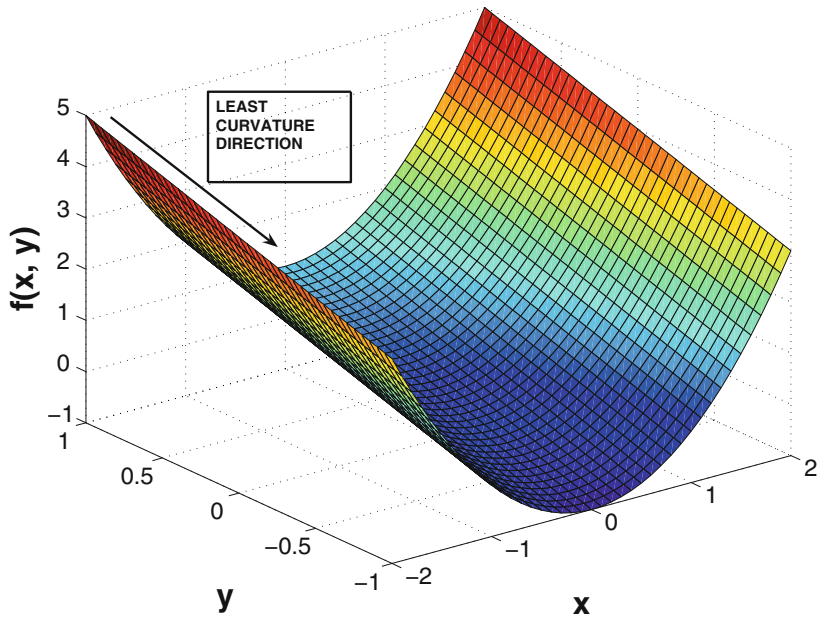


Figure 5.4: The curvature effect in valleys

The specific effect of curvature is particularly evident when one encounters loss functions in the shape of sloping or winding valleys. An example of a sloping valley is shown in Figure 5.4. A valley is a dangerous topography for a gradient-descent method, particularly if the bottom of the valley has a steep and rapidly changing surface (which creates a narrow valley). In narrow valleys, the gradient-descent method will bounce violently along the steep sides of the valley without making much progress in the gently sloping direction, where the greatest *long-term* gains are present. As we will see later in this chapter, many computational methods magnify the components of the gradient along consistent directions of movement (to discourage back-and-forth bouncing). In some cases, the steepest descent directions are

modified using such ad hoc methods, whereas in others, the curvature is explicitly used with the help of second-order derivatives. The first of these methods will be the topic of discussion in the next section.

5.3 Adjusting First-Order Derivatives for Descent

In this section, we will study computational methods that modify first-order derivatives. Implicitly, these methods do use second-order information by taking the curvature into account while modifying the components of the gradient. Many of these methods use different learning rates for different parameters. The idea is that parameters with large partial derivatives are often oscillating and zigzagging, whereas parameters with small partial derivatives tend to be more consistent but move in the same direction. These methods are also more popular than second-order methods, because they are computationally efficient to implement.

5.3.1 Momentum-Based Learning

Momentum-based methods address the issues of local optima, flat regions, and curvature-centric zigzagging by recognizing that *emphasizing medium-term to long-term directions of consistent movement* is beneficial, because they de-emphasize local distortions in the loss topology. Consequently, an aggregated measure of the feedback from previous steps is used in order to speed up the gradient-descent procedure. As an analogy, a marble that rolls down a sloped surface with many potholes and other distortions is often able to use its momentum to overcome such minor obstacles.

Consider a setting in which one is performing gradient-descent with respect to the parameter vector $\overline{W}$. The normal updates for gradient-descent with respect to the objective function J are as follows:

$$\overline{V} \Leftarrow -\alpha \frac{\partial J}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Here, α is the learning rate. We are using the matrix calculus notation $\frac{\partial J}{\partial \overline{W}}$ in lieu of ∇J. As discussed in Chapter 4, we are using the convention that the derivative of a scalar with respect to a column vector is a column vector (see page 170), which corresponds to the denominator layout in matrix calculus:

$$\nabla J = \frac{\partial J}{\partial \overline{W}} = \left[\frac{\partial J}{\partial w_1} \cdots \frac{\partial J}{\partial w_d} \right]^T$$

In momentum-based descent, the vector $\overline{V}$ inherits a fraction β of the velocity from its previous step in addition to the current gradient, where $\beta \in (0, 1)$ is the momentum parameter:

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial J}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Setting $\beta = 0$ specializes to straightforward gradient descent. Larger values of $\beta \in (0, 1)$ help the approach pick up a consistent velocity $\overline{V}$ in the correct direction. The parameter β is also referred to as the *momentum parameter* or the *friction parameter*. The word "friction" is derived from the fact that small values of β act as "brakes," much like friction.

Momentum helps the gradient descent process in navigating flat regions and local optima, such as the ones shown in Figure 5.1. A good analogy for momentum-based methods is to visualize them in a similar way as a marble rolls down a bowl. As the marble picks up

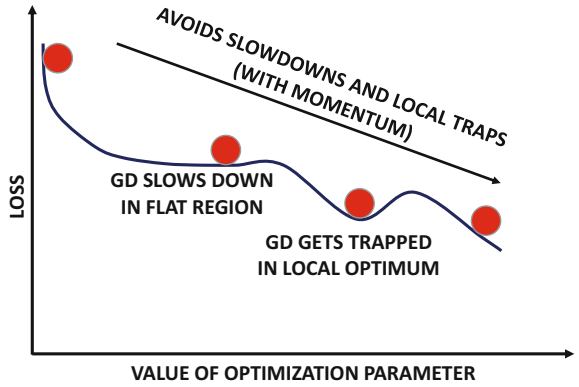


Figure 5.5: Effect of momentum in navigating complex loss surfaces. The annotation "GD" indicates pure gradient descent without momentum. Momentum helps the optimization process retain speed in flat regions of the loss surface and avoid local optima

speed, it will be able to navigate flat regions of the surface quickly and escape form local potholes in the bowl. This is because the gathered momentum helps it escape potholes. Figure 5.5, which shows a marble rolling down a complex loss surface (picking up speed as it rolls down), illustrates this concept. The use of momentum will often cause the solution to slightly overshoot in the direction where velocity is picked up, just as a marble will overshoot when it is allowed to roll down a bowl. However, with the appropriate choice of β, it will still perform better than a situation in which momentum is not used. The momentum-based method will generally perform better because the marble gains speed as it rolls down the bowl; the quicker arrival at the optimal solution more than compensates for the overshooting of the target. Overshooting is desirable to the extent that it helps avoid local optima. The parameter β controls the amount of friction that the marble encounters while rolling down the loss surface. While increased values of β help in avoiding local optima, it might also increase oscillation at the end. In this sense, the momentum-based method has a neat interpretation in terms of the physics of a marble rolling down a complex loss surface. Setting $\beta > 1$ can cause instability and divergence, because gradient descent can pick up speed in an uncontrolled way.

In addition, momentum-based methods help in reducing the undesirable effects of curvature in the loss surface of the objective function. Momentum-based techniques recognize that zigzagging is a result of highly contradictory steps that cancel out one another and reduce the *effective* size of the steps in the correct (long-term) direction. An example of this scenario is illustrated in Figure 5.2(b). Simply attempting to increase the size of the step in order to obtain greater movement in the correct direction might actually move the current solution even further away from the optimum solution. In this point of view, it makes a lot more sense to move in an "averaged" direction of the last few steps, so that the zigzagging is smoothed out. This type of averaging is achieved by using the momentum from the previous steps. Oscillating directions do not contribute consistent velocity to the update.

With momentum-based descent, the learning is accelerated, because one is generally moving in a direction that often points closer to the optimal solution and the useless "sideways" oscillations are muted. The basic idea is to give greater preference to *consistent* directions over multiple steps, which have greater importance in the descent. This allows the use of larger steps in the correct direction without causing overflows or "explosions" in the sideways direction. As a result, learning is accelerated. An example of the use of

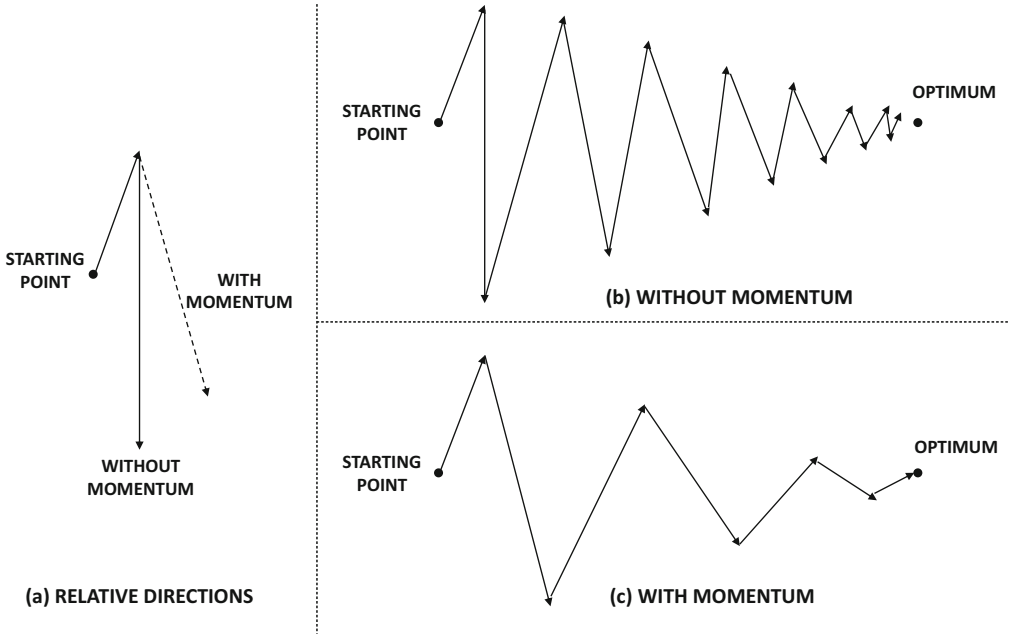


Figure 5.6: Effect of momentum in smoothing zigzag updates

momentum is illustrated in Figure 5.6. It is evident from Figure 5.6(a) that momentum increases the relative component of the gradient in the correct direction. The corresponding effects on the updates are illustrated in Figure 5.6(b) and (c). It is evident that momentum-based updates can reach the optimal solution in fewer updates. One can also understand this concept by visualizing the movement of a marble down the valley of Figure 5.4. As the marble gains speed down the gently sloping valley, the effects of bouncing along the sides of the valley will be muted over time.

5.3.2 AdaGrad

In the AdaGrad algorithm [38], one keeps track of the aggregated squared magnitude of the partial derivative with respect to each parameter over the course of the algorithm. The square-root of this value is *proportional* to the root-mean-squared slope for that parameter (although the absolute value will increase with the number of epochs because of successive aggregation).

Let A_i be the aggregate value for the ith parameter. Therefore, in each iteration, the following update is performed with respect to the objective function J:

$$A_i \Leftarrow A_i + \left(\frac{\partial J}{\partial w_i}\right)^2 ; \quad \forall i \tag{5.1}$$

The update for the ith parameter w_i is as follows:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial J}{\partial w_i}\right) ; \quad \forall i$$

If desired, one can use $\sqrt{A_i + \epsilon}$ in the denominator instead of $\sqrt{A_i}$ to avoid ill-conditioning. Here, ϵ is a small positive value such as 10^{-8}.

Scaling the derivative inversely with $\sqrt{A_i}$ is a kind of "signal-to-noise" normalization because A_i only measures the historical magnitude of the gradient rather than its sign; it encourages faster *relative* movements along gently sloping directions with consistent sign of the gradient. If the gradient component along the ith direction keeps wildly fluctuating between $+100$ and -100, this type of magnitude-centric normalization will penalize that component far more than another gradient component that consistently takes on the value in the vicinity of 0.1 (but with a consistent sign). For example, in Figure 5.6, the movements along the oscillating direction will be de-emphasized, and the movement along the consistent direction will be emphasized. However, absolute movements along all components will tend to slow down over time, which is the main problem with the approach. The slowing down is caused by the fact that A_i is the *aggregate* value of the entire history of partial derivatives. This will lead to diminishing values of the scaled derivative. As a result, the progress of AdaGrad might prematurely become too slow, and it will eventually (almost) stop making progress. Another problem is that the aggregate scaling factors depend on ancient history, which can eventually become stale. It turns out that the exponential averaging of RMSProp can address both issues.

5.3.3 RMSProp

The RMSProp algorithm [61] uses a similar motivation as AdaGrad for performing the "signal-to-noise" normalization with the absolute magnitude $\sqrt{A_i}$ of the gradients. However, instead of simply adding the squared gradients to estimate A_i, it uses *exponential averaging*. Since one uses *averaging* to normalize rather than *aggregate* values, the progress is not slowed prematurely by a constantly increasing scaling factor A_i. The basic idea is to use a decay factor $\rho \in (0, 1)$, and weight the squared partial derivatives occurring t updates ago by ρ^t. Note that this can be easily achieved by multiplying the current squared aggregate (i.e., *running* estimate) by ρ and then adding $(1 - \rho)$ times the current (squared) partial derivative. The running estimate is initialized to 0. This causes some (undesirable) bias in early iterations, which disappears over the longer term. Therefore, if A_i is the exponentially averaged value of the ith parameter w_i, we have the following way of updating A_i:

$$A_i \Leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial J}{\partial w_i} \right)^2 ; \quad \forall i \tag{5.2}$$

The square-root of this value for each parameter is used to normalize its gradient. Then, the following update is used for (global) learning rate α:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial J}{\partial w_i} \right) ; \quad \forall i$$

If desired, one can use $\sqrt{A_i + \epsilon}$ in the denominator instead of $\sqrt{A_i}$ to avoid ill-conditioning. Here, ϵ is a small positive value such as 10^{-8}. Another advantage of RMSProp over AdaGrad is that the importance of ancient (i.e., stale) gradients decays exponentially with time. The drawback of RMSProp is that the running estimate A_i of the second-order moment is biased in early iterations because it is initialized to 0.

5.3.4 Adam

The Adam algorithm uses a similar "signal-to-noise" normalization as AdaGrad and RMSProp; however, it also incorporates momentum into the update. In addition, it directly addresses the initialization bias inherent in the exponential smoothing of pure RMSProp.

As in the case of RMSProp, let A_i be the exponentially averaged value of the ith parameter w_i. This value is updated in the same way as RMSProp with the decay parameter $\rho \in (0, 1)$:

$$A_i \Leftarrow \rho A_i + (1 - \rho) \left(\frac{\partial J}{\partial w_i} \right)^2; \quad \forall i \tag{5.3}$$

At the same time, an exponentially smoothed value of the gradient is maintained for which the ith component is denoted by F_i. This smoothing is performed with a different decay parameter ρ_f:

$$F_i \Leftarrow \rho_f F_i + (1 - \rho_f) \left(\frac{\partial J}{\partial w_i} \right); \quad \forall i \tag{5.4}$$

This type of exponentially smoothing of the gradient with ρ_f is a variation of the momentum method discussed in Section 5.3.1 (which is parameterized by a friction parameter β instead of ρ_f). Then, the following update is used at learning rate α_t in the tth iteration:

$$w_i \Leftarrow w_i - \frac{\alpha_t}{\sqrt{A_i}} F_i; \quad \forall i$$

There are two key differences from the RMSProp algorithm. First, the gradient is replaced with its exponentially smoothed value in order to incorporate momentum. Second, the learning rate α_t now depends on the iteration index t, and is defined as follows:

$$\alpha_t = \alpha \underbrace{\left(\frac{\sqrt{1 - \rho^t}}{1 - \rho_f^t} \right)}_{\text{Adjust Bias}} \tag{5.5}$$

Technically, the adjustment to the learning rate is actually a bias correction factor that is applied to account for the unrealistic initialization of the two exponential smoothing mechanisms, and it is particularly important in early iterations. Both F_i and A_i are initialized to 0, which causes bias in early iterations. The two quantities are affected differently by the bias, which accounts for the ratio in Equation 5.5. It is noteworthy that each of ρ^t and ρ_f^t converge to 0 for large t because $\rho, \rho_f \in (0, 1)$. As a result, the initialization bias correction factor of Equation 5.5 converges to 1, and α_t converges to α. The default suggested values of ρ_f and ρ are 0.9 and 0.999, respectively, according to the original Adam paper [72]. Refer to [72] for details of other criteria (such as parameter sparsity) used for selecting ρ and ρ_f. Like other methods, Adam uses $\sqrt{A_i + \epsilon}$ (instead of $\sqrt{A_i}$) in the denominator of the update for better conditioning. The Adam algorithm is extremely popular because it incorporates most of the advantages of other algorithms, and often performs competitively with respect to the best of the other methods [72].

5.4 The Newton Method

The use of second-order derivatives has found a modest level of renewed popularity in recent years. Such methods can partially alleviate some of the problems caused by the high curvature of the loss function. This is because second-order derivatives encode the rate of change of the gradient in each direction, which is a more formal description of the concept of curvature. The Newton method uses a trade-off between the first- and second-order derivatives in order to descend in directions that are sufficiently steep and also do not have drastically changing gradients. Such directions allow the use of fewer steps with better

individual loss improvements. In the special case of quadratic loss functions, the Newton method requires a single step.

5.4.1 The Basic Form of the Newton Method

Consider the parameter vector $\overline{W} = [w_1 \ldots w_d]^T$ for which the second-order derivatives of the objective function $J(\overline{W})$ are of the following form:

$$H_{ij} = \frac{\partial^2 J(\overline{W})}{\partial w_i \partial w_j}$$

Note that the partial derivatives use all pairwise parameters in the denominator. Therefore, for a neural network with d parameters, we have a $d \times d$ *Hessian matrix* H, for which the (i, j)th entry is H_{ij}.

The Hessian can also be defined as the *Jacobian* of the gradient with respect to the weight vector. As discussed in Chapter 4, a Jacobian is a vector-to-vector derivative in matrix calculus, and therefore the result is a matrix. The derivative of an m-dimensional column vector with respect to an d-dimensional column vector is a $d \times m$ matrix in the denominator layout of matrix calculus, whereas it is an $m \times d$ matrix in the numerator layout (see page 170). The Jacobian is an $m \times d$ matrix, and therefore conforms to the numerator layout. In this book, we are consistently using the denominator layout, and therefore, the Jacobian of the m-dimensional vector $\overline{h}$ with respect to the d-dimensional vector $\overline{w}$ is defined as the *transpose* of the vector-to-vector derivative:

$$\text{Jacobian}(\overline{h}, \overline{w}) = \left[\frac{\partial \overline{h}}{\partial \overline{w}}\right]^T = \left[\frac{\partial h_i}{\partial w_j}\right]_{m \times d} \text{ matrix} \tag{5.6}$$

However, the transposition does not really matter in the case of the Hessian, which is symmetric. Therefore, the Hessian can also be defined as follows:

$$H = \left[\frac{\partial \nabla J(\overline{W})}{\partial \overline{W}}\right]^T = \frac{\partial \nabla J(\overline{W})}{\partial \overline{W}} \tag{5.7}$$

The Hessian can be viewed as the natural generalization of the second derivative to multivariate data. Like the univariate Taylor series expansion of the second derivative, it can be used for the multivariate Taylor-series expansion by replacing the scalar second derivative with the Hessian. Recall that the (second-order) Taylor-series expansion of a univariate function $f(w)$ about the scalar w_0 may be defined as follows (cf. Section 1.5.1 of Chapter 1):

$$f(w) \approx f(w_0) + (w - w_0) f'(w_0) + \frac{(w - w_0)^2}{2} f''(w_0) \tag{5.8}$$

It is noteworthy that the Taylor approximation is accurate when $|w - w_0|$ is small, and it starts losing its accuracy for non-quadratic functions when $|w - w_0|$ increases (as the contribution of the higher-order terms increases as well). One can also write a quadratic approximation of the *multivariate* loss function $J(\overline{W})$ in the vicinity of parameter vector $\overline{W}_0$ by using the following Taylor expansion:

$$J(\overline{W}) \approx J(\overline{W}_0) + [\overline{W} - \overline{W}_0]^T [\nabla J(\overline{W}_0)] + \frac{1}{2} [\overline{W} - \overline{W}_0]^T H [\overline{W} - \overline{W}_0] \tag{5.9}$$

As in the case of the univariate expansion, the accuracy of this approximation falls off with increasing value of $\|\overline{W} - \overline{W}_0\|$, which is the Euclidean distance between $\overline{W}$ and $\overline{W}_0$. Note that the Hessian H is computed at $\overline{W}_0$. Here, the parameter vectors $\overline{W}$ and $\overline{W}_0$ are d-dimensional column vectors. This is a quadratic approximation, and one can simply set the gradient to 0, which results in the following optimality condition for the quadratic approximation:

$$\nabla J(\overline{W}) = \overline{0}, \qquad \text{[Gradient of Loss Function]}$$
$$\nabla J(\overline{W}_0) + H[\overline{W} - \overline{W}_0] = \overline{0}, \qquad \text{[Gradient of Taylor approximation]}$$

The optimality condition above only finds a critical point, and the convexity of the function is important to ensure that this critical point is a minimum. One can rearrange the above optimality condition to obtain the following Newton update:

$$\overline{W}^* \Leftarrow \overline{W}_0 - H^{-1}[\nabla J(\overline{W}_0)] \tag{5.10}$$

One interesting characteristic of this update is that it is directly obtained from an optimality condition, and therefore there is no learning rate. In other words, this update is approximating the loss function with a quadratic bowl and moving *exactly* to the bottom of the bowl *in a single step*; the learning rate is already incorporated implicitly. Recall from Figure 5.2 that first-order methods bounce along directions of high curvature. Of course, the bottom of the quadratic approximation is not the bottom of the true loss function, and therefore multiple Newton updates will be needed. Therefore, the basic Newton method for non-quadratic functions initializes $\overline{W}$ to an initial point $\overline{W}_0$, performs the updates as follows:

1. Compute the gradient $\nabla J(\overline{W})$ and the Hessian H at the current parameter vector $\overline{W}$.
2. Perform the Newton update:

$$\overline{W} \Leftarrow \overline{W} - H^{-1}[\nabla J(\overline{W})]$$

3. If convergence has not occurred, go back to step 1.

Although the algorithm above is iterative, the Newton method requires only a single step for the special case of quadratic functions. The main difference of Equation 5.10 from the update of steepest-gradient descent is pre-multiplication of the steepest direction (which is $[\nabla J(\overline{W}_0)]$) with the inverse of the Hessian. This multiplication with the inverse Hessian plays a key role in changing the direction of the steepest-gradient descent, so that one can take larger steps in that direction (resulting in better improvement of the objective function) even if the *instantaneous* rate of change in that direction is not as large as the steepest-descent direction. This is because the Hessian encodes how fast the gradient is changing in each direction. Changing gradients are bad for larger updates because one might inadvertently worsen the objective function, if the signs of many components of the gradient change during the step. It is profitable to move in directions where the ratio of the gradient to the rate of change of the gradient is large, so that one can take larger steps while being confident that the movement is not causing unexpected changes because of the changed gradient. Pre-multiplication with the inverse of the Hessian achieves this goal. The effect of the pre-multiplication of the steepest-descent direction with the inverse Hessian is shown in Figure 5.7. It is helpful to reconcile this figure with the example of the quadratic bowl in Figure 5.2. In a sense, pre-multiplication with the inverse Hessian biases

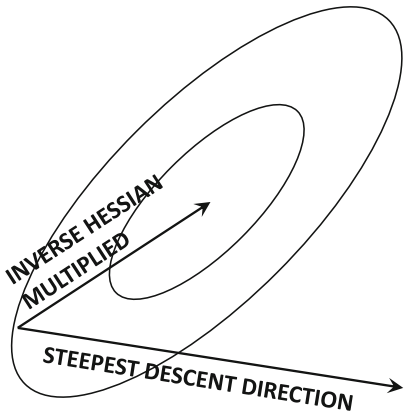


Figure 5.7: The effect of pre-multiplication of steepest-descent direction with the inverse Hessian

the learning steps towards low-curvature directions. This situation also arises in valleys like the ones shown in Figure 5.4. Multiplication with the inverse Hessian will tend to favor the gently sloping (but low curvature) direction, which is a better direction of long-term movement. Furthermore, if the Hessian is negative semi-definite at a particular point (rather than positive semi-definite), the Newton method might move in the wrong direction towards a maximum (rather than a minimum). Unlike gradient descent, the Newton method only finds critical points rather than minima.

5.4.2 Importance of Line Search for Non-quadratic Functions

It is noteworthy that the update for a non-quadratic function can be somewhat unpredictable because one moves to the bottom of a *local quadratic approximation* caused by the Taylor expansion. This local quadratic approximation can sometimes be very poor as one moves further away from the point of the Taylor approximation. Therefore, it is possible for a Newton step to worsen the quality of the objective function if one simply moves to the bottom of the local quadratic approximation. In order to understand this point, we will consider the simple case of a univariate function in Figure 5.8, where both the original function and its quadratic approximation are shown. Both the starting and ending points of a Newton step are shown, and the objective function value of the ending point differs considerably between the true function and the quadratic approximation (although the starting points are the same). As a result, the Newton step actually *worsens* the objective function value. One can view this situation in an analogous way to the problems faced by gradient descent; while gradient-descent faces problems even in quadratic functions (in terms of bouncing behavior), a "quadratically-savvy" method like the Newton technique faces problems in the case of higher-order functions.

This problem can be alleviated by exact or approximate line search, as discussed in Section 4.4.3 of Chapter 4. Line search adjusts the size of the step, so as to terminate at a better point in terms of the *true* objective function value. For example, when line search is used for the objective function in Figure 5.8, the size of the step is much smaller. It also has a much lower value of the (true) objective function. Note that line search could result in either smaller or larger steps than those computed by the vanilla Newton method.

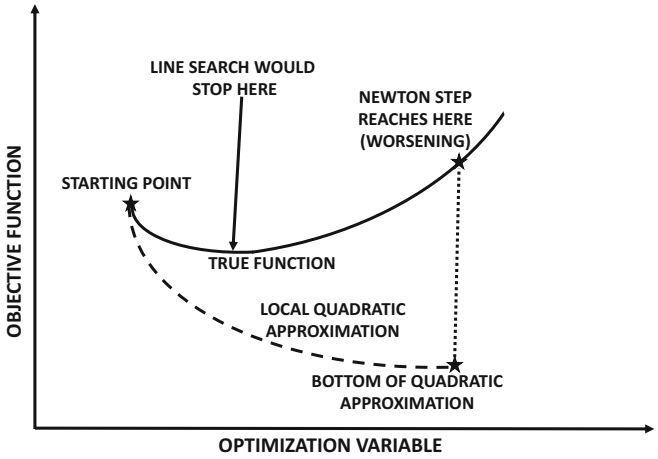


Figure 5.8: A Newton step can worsen the objective function in large steps for non-quadratic functions, because the quadratic approximation increasingly deviates from the true function. A line search can ameliorate the worsening

5.4.3 Example: Newton Method in the Quadratic Bowl

We will revisit how the Newton method behaves in the quadratic bowl of Figure 5.2. Consider the following elliptical objective function, which is the same as the one discussed in Figure 5.2(b):

$$J(w_1, w_2) = w_1^2 + 4w_2^2$$

This is a very simple convex quadratic, whose optimal point is the origin. Applying straightforward gradient descent starting at any point like $[w_1, w_2] = [1, 1]$ will result in the type of bouncing behavior shown in Figure 5.2(b). On the other hand, consider the Newton method, starting at the point $[w_1, w_2] = [1, 1]$. The gradient may be computed as $\nabla J(1, 1) = [2w_1, 8w_2]^T = [2, 8]^T$. Furthermore, the Hessian of this function is a constant that is independent of $[w_1, w_2]^T$:

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix}$$

Applying the Newton update results in the following:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \Leftarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 2 \\ 8 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

In other words, a single step suffices to reach the optimum point of this quadratic function. This is because the second-order Taylor "approximation" of a quadratic function is exact, and the Newton method solves this approximation in each iteration. Of course, real-world functions are not quadratic, and therefore multiple steps are typically needed.

5.4.4 Example: Newton Method in a Non-quadratic Function

In this section, we will modify the objective function of the previous section to make it non-quadratic. The corresponding function is as follows:

$$J(w_1, w_2) = w_1^2 + 4w_2^2 - \cos(w_1 + w_2)$$

It is assumed that w_1 and w_2 are expressed[1] in radians. Note that the optimum of this objective function is still $[w_1, w_2] = [0, 0]$, since the value of $J(0, 0)$ is -1 at this point, where each additive term of the above expression takes on its minimum value. We will again start at $[w_1, w_2] = [1, 1]$, and show that one iteration no longer suffices in this case. In this case, we can show that the gradients and Hessian are as follows:

$$\nabla J(1,1) = \begin{bmatrix} 2+\sin(2) \\ 8+\sin(2) \end{bmatrix} = \begin{bmatrix} 2.91 \\ 8.91 \end{bmatrix}$$

$$H = \begin{bmatrix} 2+\cos(2) & \cos(2) \\ \cos(2) & 8+\cos(2) \end{bmatrix} = \begin{bmatrix} 1.584 & -0.416 \\ -0.416 & 7.584 \end{bmatrix}$$

The inverse of the Hessian is as follows:

$$H^{-1} = \begin{bmatrix} 0.64 & 0.035 \\ 0.035 & 0.134 \end{bmatrix}$$

Therefore, we obtain the following Newton update:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \Leftarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.64 & 0.035 \\ 0.035 & 0.134 \end{bmatrix} \begin{bmatrix} 2.91 \\ 8.91 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 2.1745 \\ 1.296 \end{bmatrix} = \begin{bmatrix} -1.1745 \\ -0.2958 \end{bmatrix}$$

Note that we do reach closer to an optimal solution, although we certainly do not reach the optimum point. This is because the objective function is not quadratic in this case, and one is only reaching the bottom of the *approximate* quadratic bowl of the objective function. However, Newton's method does find a better point in terms of the true objective function value. The approximate nature of the Hessian is why one must use either exact or approximate line search to control the step size. Note that if we used a step-size of 0.6 instead of the default value of 1, one would obtain the following solution:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \Leftarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.6 \begin{bmatrix} 2.1745 \\ 1.296 \end{bmatrix} = \begin{bmatrix} -0.30 \\ 0.22 \end{bmatrix}$$

Although this is only a very rough approximation to the optimal step size, it still reaches much closer to the true optimal value of $[w_1, w_2] = [0, 0]$. It is also relatively easy to show that this set of parameters yields a much better objective function value. This step would need to be repeated in order to reach closer and closer to an optimal solution.

5.5 Newton Methods in Machine Learning

In this section, we will provide some examples of the use of the Newton method for machine learning.

5.5.1 Newton Method for Linear Regression

We will start with the linear-regression loss function. Even though linear regression is relatively easy to solve with first-order methods, the approach is instructive because it allows us to relate the Newton method to the most straightforward closed-form solution of linear regression (cf. Section 4.7 of Chapter 4). The objective function of linear regression for an

[1] This ensures simplicity, as all calculus operations assume that angles are expressed in radians.

$n \times d$ data matrix D, n-dimensional column vector of target variables $\overline{y}$, and d-dimensional column vector $\overline{W}$ of parameters, is as follows:

$$J(\overline{W}) = \frac{1}{2}\|D\overline{W} - \overline{y}\|^2 = \frac{1}{2}[D\overline{W} - \overline{y}]^T[D\overline{W} - \overline{y}] \tag{5.11}$$

The Newton method requires us to compute both the gradient and the Hessian. We will start by computing the gradient, and then compute the Jacobian of the gradient in order to compute the Hessian. The loss function can be expanded as $\overline{W}^T D^T D\overline{W}/2 - \overline{y}^T D\overline{W} + \overline{y}^T\overline{y}/2$. We can use identities (i) and (ii) from Table 4.2(a) of Chapter 4 to compute the gradients of the individual terms. Therefore, we obtain the gradient of the loss function as follows:

$$\nabla J(\overline{W}) = D^T D\overline{W} - D^T\overline{y} \tag{5.12}$$

The Hessian is obtained by computing the Jacobian of this gradient. The second term of the gradient is a constant and therefore further differentiating it will yield 0; we need only differentiate the first term. On computing the vector-to-vector derivative of the first term of the gradient with respect to $\overline{W}$, we obtain the fact that the Hessian is $D^T D$. This observation can be verified directly using the matrix calculus identity (i) of Table 4.2(b) in Chapter 4. We summarize this observation as follows:

Observation 5.5.1 (Hessian of Squared Loss) *Let $J(\overline{W}) = \frac{1}{2}\|D\overline{W} - \overline{y}\|^2$ be the loss function of linear regression for an $n \times d$ data matrix D, a d-dimensional column vector $\overline{W}$ of coefficients and n-dimensional column vector $\overline{y}$ of targets. Then, the Hessian of the loss function is given by $D^T D$.*

It is also helpful to view the Hessian as the sum of point-specific Hessians, since the Hessian of any linearly additive function is the sum of the Hessians of the individual terms:

Observation 5.5.2 (Point-Specific Hessian of Squared Loss) *Let $J_i = \frac{1}{2}(\overline{W} \cdot \overline{X}_i - y_i)^2$ be the loss function of linear regression for a single training pair $(\overline{X}_i, y_i)$. Then, the point specific Hessian of the squared loss of J_i is given by the outer-product $\overline{X}_i^T \overline{X}_i$.*

Note that $D^T D$ is simply the sum over all $\overline{X}_i^T \overline{X}_i$, since any matrix multiplication can be decomposed into the sum of outer-products (Lemma 1.2.1 of Chapter 1):

$$D^T D = \sum_{i=1}^{n} \overline{X}_i^T \overline{X}_i$$

This is consistent with the fact that Hessian of the full data-specific loss function is the sum of the point-specific Hessians.

One can now combine the Hessian and gradient to obtain the Newton update. A neat result is that the Newton update for least-squares regression and classification simplifies to the closed-form solution of linear regression result discussed in Chapter 4. Given the current vector $\overline{W}$, the Newton update is as follows (based on Equation 5.10):

$$\begin{aligned}\overline{W} &\Leftarrow \overline{W} - H^{-1}[\nabla J(\overline{W})] = \overline{W} - (D^T D)^{-1}[D^T D\overline{W} - D^T\overline{y}] \\ &= \underbrace{\overline{W} - \overline{W}}_{0} + (D^T D)^{-1} D^T\overline{y} = (D^T D)^{-1} D^T\overline{y}\end{aligned}$$

Note that the right-hand side is free of $\overline{W}$, and therefore we need a single "update" step in closed form. This solution is identical to Equation 4.39 of Chapter 4! This equivalence

is not surprising. The closed-form solution of Chapter 4 is obtained by setting the gradient of the loss function to 0. The Newton method also sets the gradient of the loss function to 0 after representing it using a second-order Taylor expansion (which is exact for quadratic functions).

Problem 5.5.1 *Derive the Newton update for least-squares regression, when Tikhonov regularization with parameter $\lambda > 0$ is used. Show that the final solution is $\overline{W}^* = (D^T D + \lambda I)^{-1} D^T \overline{y}$, which is the same regularized solution derived in Chapter 4.*

5.5.2 Newton Method for Support-Vector Machines

Next, we will discuss the case of the support vector machine with binary class variables $\overline{y} = [y_1, \ldots, y_n]^T$, where each $y_i \in \{-1, +1\}$. All other notations, such as D, $\overline{W}$, and $\overline{X}_i$ are the same as those of the previous section. The use of the hinge-loss is not common with the Newton method because of its non-differentiability at specific points. Although the non-differentiability does not cause too many problems for straightforward gradient descent (see Section 4.8.2 of Chapter 4), it becomes a bigger problem when dealing with second-order methods. Although one can create a differentiable *Huber loss* approximation [28], we will only discuss the L_2-SVM here. One can write its objective function in terms of the rows of matrix D, which are $\overline{X}_1 \ldots \overline{X}_n$, and the elements of $\overline{y}$, which are $y_1 \ldots y_n$:

$$J(\overline{W}) = \frac{1}{2} \sum_{i=1}^{n} \max\left\{0, \left(1 - y_i[\overline{W} \cdot \overline{X}_i^T]\right)\right\}^2 \quad [L_2\text{-loss SVM}]$$

We have omitted the regularization term for simplicity. This loss can be decomposed as $J(\overline{W}) = \sum_i J_i$, where J_i is the point-specific loss. The point-specific loss for the ith point can be expressed in a form corresponding to identity (v) of Table 4.2(a) in Chapter 4:

$$J_i = f_i(\overline{W} \cdot \overline{X}_i^T) = \frac{1}{2} \max\left\{0, \left(1 - y_i[\overline{W} \cdot \overline{X}_i^T]\right)\right\}^2$$

Note the use of the function $f_i(\cdot)$ in the above expression, which is defined for L_2-loss SVMs as follows:

$$f_i(z) = \frac{1}{2} \max\{0, 1 - y_i z\}^2$$

This function will eventually need to be differentiated during gradient descent:

$$\frac{\partial f_i(z)}{\partial z} = f_i'(z) = -y_i \max\{0, 1 - y_i z\}$$

Therefore, we have $J_i = f_i(z_i)$, where $z_i = \overline{W} \cdot \overline{X}_i^T$. The derivative of $J_i = f_i(z_i)$ with respect to $\overline{W}$ is computed using the chain rule:

$$\frac{\partial J_i}{\partial \overline{W}} = \frac{\partial f_i(z_i)}{\partial \overline{W}} = \frac{\partial f_i(z_i)}{\partial z_i} \underbrace{\frac{\partial z_i}{\partial \overline{W}}}_{\overline{X}_i^T} = -y_i \max\{0, 1 - y_i(\overline{W} \cdot \overline{X}_i)\} \overline{X}_i^T \tag{5.13}$$

Note that this derivative is in the same form as identity (v) of Table 4.2(a). In order to compare the gradients of least-squares classification and the L_2-SVM, we restate them next to each other:

$$\frac{\partial J_i}{\partial \overline{W}} = -y_i(1 - y_i(\overline{W} \cdot \overline{X}_i^T))\overline{X}_i^T \qquad \text{[Least-Squares Classification]}$$
$$\frac{\partial J_i}{\partial \overline{W}} = -y_i \max\{0, 1 - y_i(\overline{W} \cdot \overline{X}_i^T)\}\overline{X}_i^T \quad [L_2\text{-SVM}]$$

The least-squares classification and the L_2-SVM have a similar gradient, except that the contributions of instances that are correctly classified in a confident way (i.e., instances satisfying $y_i(\overline{W} \cdot \overline{X}_i^T) \geq 1$) are not included in the SVM. One can use $y_i^2 = 1$ to rewrite the gradient of the L_2-SVM in terms of the indicator function as follows:

$$\frac{\partial J_i}{\partial \overline{W}} = \underbrace{(\overline{W} \cdot \overline{X}_i^T - y_i) I([1 - y_i(\overline{W} \cdot \overline{X}_i^T)] > 0)}_{\text{scalar}} \underbrace{\overline{X}_i^T}_{\text{vector}} \quad [L_2\text{-SVM}]$$

The binary indicator function $I(\cdot)$ takes on the value of 1 when the condition inside it is satisfied. Therefore, the overall gradient of $J(\overline{W})$ with respect to $\overline{W}$ can be written as follows:

$$\nabla J(\overline{W}) = \sum_{i=1}^{n} \frac{\partial J_i}{\partial \overline{W}} = \sum_{i=1}^{n} \underbrace{(\overline{W} \cdot \overline{X}_i^T - y_i) I([1 - y_i(\overline{W} \cdot \overline{X}_i^T)] > 0)}_{\text{scalar}} \underbrace{\overline{X}_i^T}_{\text{vector}}$$
$$= D^T \Delta_w (D\overline{W} - \overline{y})$$

Here, Δ_w is an $n \times n$ diagonal matrix in which the (i, i)th entry contains the indicator function $I([1 - y_i(\overline{W} \cdot \overline{X}_i^T)] > 0)$ for the ith training instance.

Next, we focus on the computation of the Hessian. We would first like to compute the Jacobian of the point-specific gradient $\frac{\partial J_i}{\partial \overline{W}}$ in order to compute the point-specific Hessian, and then add up the point-specific Hessians. In important point is that the gradient is the product of a scalar $s = -y_i \max\{0, 1 - y_i(\overline{W} \cdot \overline{X}_i)\}$ (dependent on $\overline{W}$) and the vector $\overline{X}_i^T$ (independent of $\overline{W}$). This fact simplifies the computation of the *point-specific* Hessian H_i (i.e., transposed vector derivative of the gradient), using the product-of-variables identity in Table 4.2(b):

$$H_i = \overline{X}_i^T \left[\frac{\partial s}{\partial \overline{W}}\right]^T = \overline{X}_i^T \left[y_i^2 I([1 - y_i(\overline{W} \cdot \overline{X}_i^T)] > 0)\overline{X}_i\right]$$
$$= I([1 - y_i(\overline{W} \cdot \overline{X}_i^T)] > 0)[\overline{X}_i^T \overline{X}_i] \quad [\text{Setting } y_i^2 = 1]$$

The overall Hessian H is the sum of the point-specific Hessians:

$$H = \sum_{i=1}^{n} H_i = \sum_{i=1}^{n} \underbrace{I([1 - y_i(\overline{W} \cdot \overline{X}_i^T)] > 0)}_{\text{Binary Indicator}} \underbrace{[\overline{X}_i^T \overline{X}_i]}_{\text{Outer Prod.}}$$

How is the Hessian of the L_2-SVM different from that in least-squares classification? Note that the Hessian of least-squares classification can be written as the sum of outer products $\sum_i [\overline{X}_i^T \overline{X}_i]$ of the individual points. The Hessian of the L_2-SVM also sums the outer products, except that it uses an indicator function to drop out the points that meet the margin condition (of being classified correctly with sufficient margin). Such points do not contribute to the Hessian. Therefore, one can write the Hessian of the L_2-SVM loss as follows:

$$H = D^T \Delta_w D$$

Here, Δ_w is the same $n \times n$ binary diagonal matrix Δ_w that is used in the expression for the gradient. The value of Δ_w will change over time during learning, as different training instances move in and out of correct classification and therefore contribute in varying ways to Δ_w. The key point is that rows drop in and out in terms of their contributions to the gradient and the Hessian, as $\overline{W}$ changes. This is the reason that we have subscripted Δ with w to indicate that it depends on the parameter vector.

Therefore, at any given value of the parameter vector, the Newton update of the L_2-loss SVM is as follows:

$$\begin{aligned}\overline{W} &\Leftarrow \overline{W} - H^{-1}[\nabla J(\overline{W})] = \overline{W} - (D^T \Delta_w D)^{-1}[D^T \Delta_w (D\overline{W} - \overline{y})] \\ &= \underbrace{\overline{W} - \overline{W}}_{0} + (D^T \Delta_w D)^{-1} D^T \Delta_w \overline{y} = (D^T \Delta_w D)^{-1} D^T \Delta_w \overline{y}\end{aligned}$$

This form is almost identical to least-squares classification, except that we are dropping the instances that are correctly classified in a strong way. At first glance, it might seem that the L_2-SVM also requires a single iteration like least-squares regression, because the vector $\overline{W}$ has disappeared on the right-hand side. However, this does not mean that the right-hand side is independent of $\overline{W}$. The matrix Δ_w *does* depend on the weight vector, and will change once $\overline{W}$ is updated. Therefore, one must recompute Δ_w in each iteration and repeat the above step to convergence.

The second point is that *line search becomes important in each update of the L_2-SVM, as we are no longer dealing with a quadratic function.* Therefore, we can add line search to compute the learning rate α_t in the tth iteration. This results in the following update:

$$\begin{aligned}\overline{W} &\Leftarrow \overline{W} - \alpha_t (D^T \Delta_w D)^{-1}[D^T \Delta_w D\overline{W} - D_w^T \Delta_w \overline{y}] \\ &= \overline{W}(1 - \alpha_t) + \alpha_t (D^T \Delta_w D)^{-1} D^T \overline{y}\end{aligned}$$

Note that it is possible for line search to obtain a value of $\alpha_t > 1$, and therefore the coefficient $(1 - \alpha_t)$ of the first term can be negative. One can also derive a form of the update for the regularized SVM. We leave this problem as a practice exercise.

Problem 5.5.2 *Derive the Newton update without line-search for the L_2-SVM, when Tikhonov regularization with parameter $\lambda > 0$ is used. Show that the* **iterative update** *of the Newton method is $\overline{W} \Leftarrow (D^T \Delta_w D + \lambda I)^{-1} D^T \Delta_w \overline{y}$. All notations are the same as those used for the L_2-SVM in this section.*

It is noteworthy that the Newton's update uses the quadratic Taylor expansion of the non-quadratic objective function of the L_2-SVM; the second-order Taylor expansion is, therefore, only an approximation. On the other hand, least-squares regression already has a quadratic objective function, and its second-order Taylor approximation is exact. This point of view is critical in understanding why certain objective functions like least-squares regression require a single Newton update, whereas others like the SVM do not.

Problem 5.5.3 *Discuss why the Hessian is more likely to become singular towards the end of learning in the Newton method for the L_2-SVM. How would you address the problem caused by the non-invertibility of the Hessian? Also discuss the importance of line search in these cases.*

5.5.3 Newton Method for Logistic Regression

We revisit logistic regression (cf. Section 4.8.3 of Chapter 4) with training pairs $(\overline{X}_i, y_i)$. Here, each $\overline{X}_i$ is a d-dimensional row vector and $y_i \in \{-1, +1\}$. There are a total of n

training pairs, and therefore stacking up all the d-dimensional rows results in an $n \times d$ matrix D. The resulting loss function (cf. Section 4.8.3) is as follows:

$$J(\overline{W}) = \sum_{i=1}^{n} \log(1 + \exp(-y_i[\overline{W} \cdot \overline{X}_i^T]))$$

We start by defining a function for logistic loss in order to enable the (eventual) use of the chain rule:

$$f_i(z) = \log(1 + \exp(-y_i z)) \tag{5.14}$$

When z_i is set to $\overline{W} \cdot \overline{X}_i^T$, the function $f_i(z_i)$ contains the loss for the ith training point. The derivative of $f_i(z_i)$ is as follows:

$$\frac{\partial f_i(z_i)}{\partial z_i} = -y_i \frac{\exp(-y_i z_i)}{1 + \exp(-y_i z_i)} = -y_i \underbrace{\frac{1}{1 + \exp(y_i z_i)}}_{p_i}$$

The quantity $p_i = 1/(1 + \exp(y_i z_i)$ in the above expression is always interpreted as the probability of the model to make[2] a mistake, when $z_i = \overline{W} \cdot \overline{X}_i^T$. Therefore, one can express the derivative of $f_i(z_i)$ as follows:

$$\frac{\partial f_i(z_i)}{\partial z_i} = -y_i p_i$$

With this machinery and notations, one can write the objective function of logistic regression in terms of the individual losses:

$$J(\overline{W}) = \sum_{i=1}^{n} f_i(\overline{W} \cdot \overline{X}_i^T) = \sum_{i=1}^{n} f_i(z_i)$$

Then, one can compute the gradient of the loss function using the chain rule as follows:

$$\nabla J(\overline{W}) = \sum_{i=1}^{n} \underbrace{\frac{\partial f_i(z_i)}{\partial z_i}}_{-y_i p_i} \underbrace{\frac{\partial z_i}{\partial \overline{W}}}_{\overline{X}_i^T} = -\sum_{i=1}^{n} y_i p_i \overline{X}_i^T \tag{5.15}$$

The derivative of $z_i = \overline{W} \cdot \overline{X}_i^T$ with respect to $\overline{W}$ is based on identity (v) of Table 4.2(a). To represent the gradient compactly using matrices, one can introduce an $n \times n$ diagonal matrix Δ_w^p, in which the ith diagonal entry contains the probability p_i:

$$\nabla J(\overline{W}) = -D^T \Delta_w^p \overline{y} \tag{5.16}$$

One can view Δ_w^p as a soft version of the binary matrix Δ_w used for the L_2-SVM. Therefore, we have added the superscript p to the matrix Δ_w^p in order to indicate that it is a probabilistic matrix.

The Hessian is given by the Jacobian of the gradient:

$$H = \left[\frac{\partial \nabla J(\overline{W})}{\partial \overline{W}}\right]^T = -\sum_{i=1}^{n} \left[\frac{\partial [y_i p_i \overline{X}_i^T]}{\partial \overline{W}}\right]^T = -\sum_{i=1}^{n} y_i \left[\frac{\partial [p_i \overline{X}_i^T]}{\partial \overline{W}}\right]^T \tag{5.17}$$

[2]This conclusion follows from the modeling assumption in logistic regression that the probability of a correct prediction is $p_i' = 1/(1 + \exp(-y_i z_i))$. It can be easily shown that $p_i + p_i' = 1$.

The vector $\overline{X}_i$ is independent of $\overline{W}$, whereas p_i is a scalar that depends on $\overline{W}$. In the denominator layout, the derivative of the column vector $p_i\overline{X}_i^T$ with respect to the column vector $\overline{W}$ is the matrix $\frac{\partial p_i}{\partial \overline{W}}\overline{X}_i$ based on identity (iii) of Table 4.2(b). Therefore, the Hessian can be written in matrix calculus notation as $H = -\sum_i y_i \left[\frac{\partial p_i}{\partial \overline{W}}\overline{X}_i\right]^T$. The gradient of p_i with respect to $\overline{W}$ can be computed using the chain rule with respect to intermediate variable $z_i = \overline{W} \cdot \overline{X}_i^T$ as follows:

$$\frac{\partial p_i}{\partial \overline{W}} = \frac{\partial p_i}{\partial z_i}\frac{\partial z_i}{\partial \overline{W}} = \frac{\partial p_i}{\partial z_i}\overline{X}_i^T = -\frac{y_i \exp(y_i z_i)}{(1+\exp(y_i z_i))^2}\overline{X}_i^T = -y_i p_i(1-p_i)\overline{X}_i^T \tag{5.18}$$

Substituting the gradient of p_i from Equation 5.18 in the expression $H = -\sum_i y_i \left[\frac{\partial p_i}{\partial \overline{W}}\overline{X}_i\right]^T$, we obtain the following:

$$H = \sum_i \underbrace{y_i^2}_{=1} p_i(1-p_i)\overline{X}_i^T\overline{X}_i \tag{5.19}$$

Now observe that this form is the weighted sum of matrices, where each matrix is the outer-product between a vector and itself. This form is also used in the spectral decomposition of matrices (cf. Equation 3.43 of Chapter 3), in which the weighting is handled by a diagonal matrix. Consequently, we can convert the Hessian to a form using the data matrix D as follows:

$$H = D^T \Lambda_w^u D \tag{5.20}$$

Here, Λ_w^u is a diagonal matrix of *uncertainties* in which the ith diagonal entry is simply $p_i(1-p_i)$, where p_i is the probability of making a mistake on the ith training instance with weight vector $\overline{W}$. When a point is classified with probability close to 0 or 1, the value of p_i will always be closer to 0. On the other hand, if the model is unsure about the class label of p_i, its probability will be high. Note that Λ_w^u depends on the value of the parameter vector, and we have added the notations w, u to it in order to emphasize that it is an uncertainty matrix that depends on the parameter vector. It is helpful to note that the Hessian of logistic regression is similar in form to the Hessian $D^T D$ in the "parent problem" of linear regression and the Hessian $D^T \Delta_w D$ in the L_2-SVM. The L_2-SVM explicitly drops rows that are *correctly* classified in a confident way, whereas logistic regression gives each row a soft weight depending on the level of *uncertainty* (rather than correctness) in classification.

One can now derive an expression for the Newton update for logistic regression by plugging in the expressions for the Hessian and the gradient. At any given value of the parameter vector $\overline{W}$, the update is as follows:

$$\overline{W} \Leftarrow \overline{W} + (D^T \Lambda_w^u D)^{-1} D^T \Delta_w^p \overline{y}$$

This iterative update needs to be executed to convergence. Note that Δ_w^p simply weights each class label from $\{-1, +1\}$ by the probability of making a mistake for that training instance. Therefore, instances with larger mistake probabilities are emphasized in the update. This is also an important difference from the L_2-SVM where only incorrect or marginally classified instances are used, and other "confidently correct" instances are discarded. Furthermore, the update of logistic regression uses the "uncertainty weight" in the matrix Λ_w^u. Finally, it is common to use line search in conjunction with learning rate α in order to modify the aforementioned update to the following:

$$\overline{W} \Leftarrow \overline{W} + \alpha(D^T \Lambda_w^u D)^{-1} D^T \Delta_w^p \overline{y}$$

Problem 5.5.4 *Derive the Newton update for logistic regression, when Tikhonov regularization with parameter λ is used. Show that the update is modified to the following:*

$$\overline{W} \Leftarrow \overline{W} + \alpha(D^T \Lambda_w^u D + \lambda I)^{-1}\{[D^T \Delta_w^p \overline{y}] - \lambda \overline{W}\}$$

The notations here are the same as those in the discussion of this section.

5.5.4 Connections Among Different Models and Unified Framework

The Newton update for the different models, corresponding to least-squares regression, the L_2-SVM, and logistic regression are closely related. This is not particularly surprising, since their loss functions are closely related (cf. Figure 4.9 of Chapter 4). In the following table, we list all the updates for the various Newton Methods, so that they can be compared:

Method	Update (no line search)	Update (with line search)
Linear regression and classification	$\overline{W} = (D^T D)^{-1} D^T \overline{y}$ (single step: no iterations)	Line search not needed (single step: no iterations)
L_2-SVM	$\overline{W} \Leftarrow (D^T \Delta_w D)^{-1} D^T \Delta_w \overline{y}$ (Δ_w is binary diagonal matrix) (Δ_w excludes selected points)	$\overline{W} \Leftarrow (1-\alpha_t)\overline{W} + \alpha_t (D^T \Delta_w D)^{-1} D^T \Delta_w \overline{y}$ (Δ_w is binary diagonal matrix) (Δ_w excludes selected points)
Logistic regression	$\overline{W} \Leftarrow \overline{W} + (D^T \Lambda_w^u D)^{-1} D^T \Delta_w^p \overline{y}$ (Λ_w^u, Δ_w^p are soft diagonal matrices) (Matrices use soft weights)	$\overline{W} \Leftarrow \overline{W} + \alpha_t (D^T \Lambda_w^u D)^{-1} D^T \Lambda_w^p \overline{y}$ (Λ_w^u, Δ_w^p are soft diagonal matrices) (Matrices use soft weights)

It is evident that all the updates are very similar. One can explain these differences in terms of the similarities and differences of the loss functions. For example, when the L_2-SVM is compared to least-squares classification, it is primarily different in terms of assuming zero loss for points that are classified correctly in a sufficiently "confident" way (i.e., meet the margin requirement). Similarly, when we compare the Hessian and the gradient used in the case of the L_2-SVM to that used in least-squares classification, a *binary* diagonal matrix Δ_w is used to remove the effect of these correctly classified points (whereas least-squares classification includes these points as well). The impact of changing the loss function is more complex in the case of logistic regression; points that are correctly classified with high probability are de-emphasized in the gradient, and points that the model is certain about (whether correct or incorrect) are de-emphasized in the Hessian. Furthermore, unlike the L_2-SVM, logistic regression uses soft weighting rather than hard weighting. All these connections are naturally related to the connections among their loss functions (cf. Figure 4.9 of Chapter 4). The logistic regression update is considered a soft and iterative version of the closed-form solution to least-squares regression — as a result, the Newton method for logistic regression is sometimes also referred to as the *iteratively re-weighted least-squares algorithm*.

One can also understand all these updates in the context of a unified framework, where the regularized loss function for many machine learning models can be expressed as follows:

$$J = \sum_{i=1}^{n} f_i(\overline{W} \cdot \overline{X}_i^T) + \frac{\lambda}{2}\|\overline{W}\|^2$$

Note that each $f_i(\cdot)$ also uses the observed value y_i to compute the loss, and can also be written as $L(y_i, \overline{W} \cdot \overline{X}_i^T)$. All the updates can be written in a single unified form as discussed in the result below:

Lemma 5.5.1 (Unified Newton Update for Machine Learning) *Let the objective function for a machine learning problem with d-dimensional parameter vector $\overline{W}$, and $n \times d$ data matrix D containing rows (feature vectors) $\overline{X}_1 \ldots \overline{X}_n$ be as follows:*

$$J = \sum_{i=1}^{n} L(y_i, \overline{W} \cdot \overline{X}_i^T) + \frac{\lambda}{2} \|\overline{W}\|^2$$

Here, $\overline{y} = [y_1 \ldots y_n]^T$ is the observed dependent variable parameter vector for matrix D. Then, the regularized Newton update can be written in the following form:

$$\overline{W} \Leftarrow \overline{W} - \alpha (D^T \Delta_2 D + \lambda I)^{-1} (D^T \Delta_1 \overline{1} + \lambda \overline{W})$$

Here Δ_2 is an $n \times n$ diagonal matrix whose diagonal entries contain the second derivative $L''(y_i, z_i)$ [with respect to $z_i = \overline{W} \cdot \overline{X}_i^T$] evaluated at each $(\overline{X}_i, y_i)$, and Δ_1 is an $n \times n$ diagonal matrix whose diagonal entries contain the corresponding first derivative $L'(y_i, z_i)$ evaluated at each $(\overline{X}_i, y_i)$.

We leave the proof of this lemma as an exercise for the reader (see Exercise 14).

5.6 Newton Method: Challenges and Solutions

Although the Newton method avoids many of the problems associated with gradient descent, it comes with its own set of challenges, which will be studied in this section.

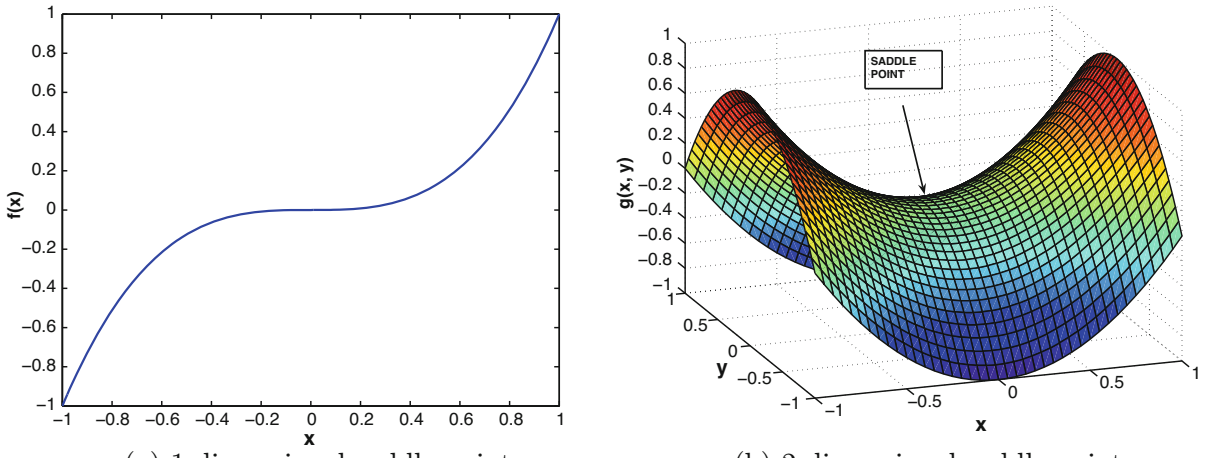
5.6.1 Singular and Indefinite Hessian

Newton's method is inherently designed for convex quadratic functions with positive-definite Hessians. The Hessian can sometimes be singular or indefinite. For example, in the case of the (unregularized) L_2-SVM, the Hessian is the (signed) sum of outer products $\overline{X}_i \overline{X}_i^T$ of points that are marginally correct or incorrect in terms of prediction. Each of these point-specific Hessians is a rank-1 matrix. We need at least d of them in order to create a $d \times d$ Hessian of full rank d (cf. Lemma 2.6.2 of Chapter 2). This might not occur near convergence.

When the Hessian is not invertible, one can either add λI to the Hessian (for regularization) or work with the pseudoinverse of the Hessian. Regularization can also convert an indefinite Hessian to a positive definite matrix by using a large enough value of λ. In particular, choosing λ to be slightly greater than the absolute value of the most negative eigenvalue (of the Hessian) will result in a positive definite Hessian. It is noteworthy that ill-conditioning problems continue to arise even with regularization (cf. Sections 2.9 and 7.4.4.1), when the Hessian is nearly singular.

5.6.2 The Saddle-Point Problem

So far, we have looked at the performance of the Newton method with convex functions. Non-convex functions bring other types of challenges such as *saddle points*. Saddle points occur when the Hessian of the loss function is indefinite. A saddle point is a stationary point



(a) 1-dimensional saddle point (b) 2-dimensional saddle point

Figure 5.9: Illustration of saddle points

(i.e., a critical point) of a gradient-descent method because its gradient is zero, but it is not a minimum (or maximum). A saddle point is an *inflection point*, which appears to be either a minimum or a maximum depending on which direction we approach it from. Therefore, the quadratic approximation of the Newton method will result in vastly different shapes depending on the precise location of current parameter vector with respect to a nearby saddle point. A 1-dimensional function with a saddle point is the following:

$$f(x) = x^3$$

This function is shown in Figure 5.9(a), and it has an inflection point at $x = 0$. Note that a quadratic approximation at $x > 0$ will look like an upright bowl, whereas a quadratic approximation at $x < 0$ will look like an inverted bowl. The second-order Taylor approximations at $x = 1$ and $x = -1$ are as follows:

$$F(x) = 1 + 3(x-1) + \frac{6(x-1)^2}{2} = 3x^2 - 3x + 1 \quad [\text{At } x = 1]$$

$$G(x) = -1 + 3(x+1) - \frac{6(x+1)^2}{2} = -3x^2 - 3x - 1 \quad [\text{At } x = -1]$$

It is not difficult to verify that one of these functions is an upright bowl (convex function) with a minimum and no maximum, whereas another is an inverted bowl (concave function) with a maximum and no minimum. Therefore, the Newton optimization will behave in an unpredictable way, depending on the current value of the parameter vector. Furthermore, even if one reaches $x = 0$ in the optimization process, both the second derivative and the first derivative will be zero. Therefore, a Newton update will take the 0/0 form and become indefinite. Such a point is a degenerate point from the perspective of numerical optimization. In general, a degenerate critical point is one where the Hessian is singular (along with the first-order condition that the gradient is zero). The problem is complicated by the fact that a degenerate critical point can be either a true optimum or a saddle point. For example, the function $h(x) = x^4$ has a degenerate critical point at $x = 0$ in which both first-order and second-order derivatives are 0. However, the point $x = 0$ is a true minimum.

It is also instructive to examine the case of a saddle point in a multivariate function, where the Hessian is not singular. An example of a 2-dimensional function with a saddle point is as follows:

$$g(x, y) = x^2 - y^2$$

This function is shown in Figure 5.9(b). The saddle point is $(0, 0)$. The Hessian of this function is as follows:

$$H = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$$

It is easy to see that the shape of this function resembles a riding saddle. In this case, approaching from the x direction or from the y direction will result in very different quadratic approximations. In one case, the function will appear to be a minimum, and in another case, the function will appear to be a maximum. Furthermore, the saddle point $[0, 0]$ will be a stationary point from the perspective of a Newton update, even though it is not an extremum. Saddle points occur frequently in regions between two hills of the loss function, and they present a problematic topography for the Newton method. Interestingly, straightforward gradient-descent methods are often able to escape from saddle points [54], because they are simply not attracted by such points. On the other hand, Newton's method is indiscriminately attracted to all critical points (such as maxima or saddle points). High-dimensional objective functions seem to contain a large number of saddle points compared to true optima (see Exercise 14). The Newton method does not always perform better than gradient descent, and the specific topography of a particular loss function may have an important role to play. The Newton method is needed for loss functions with complex curvatures, but without too many saddle points. Note that the pairing of computational algorithms (like Adam) with gradient-descent methods already changes the steepest direction in a way that incorporates several advantages of second-order methods in an implicit way. Therefore, real-world practitioners often prefer gradient-descent methods in combination with computational algorithms like Adam. Recently, some methods have been proposed [32] to address saddle points in second-order methods.

5.6.3 Convergence Problems and Solutions with Non-quadratic Functions

The first-order gradient-descent method works well with the SVM and logistic regression, because these are convex functions. In such cases, gradient descent is almost always guaranteed to converge to an optimum, as long as step-sizes are chosen appropriately. However, a surprising fact is that the (more sophisticated) Newton method is not guaranteed to converge to an optimal solution. Furthermore, one is not even guaranteed to improve the objective function value with a given update, if one uses the most basic form of the Newton method.

Here, it is important to understand that the Newton method uses a local Taylor approximation at the current parameter vector $\overline{w}$ to compute both the gradient and the Hessian; if the quadratic approximation deteriorates rapidly with increasing distance from the parameter vector $\overline{W}$, the results can be uncertain. Just as first-order gradient descent uses the instantaneous direction of steepest descent as an approximation, the second-order method uses a local Taylor approximation which is correct only over an infinitesimal region of the space. As one makes steps of larger size, the effect of the step can be uncertain.

In order to understand this point, let us examine a simple 1-dimensional classification problem in which the feature-label pairs are $(1, 1)$, $(2, 1)$, and $(3, -1)$. We have a single parameter w that needs to be learned. The objective function of least-squares classification is as follows:

$$J = (1 - w)^2 + (1 - 2w)^2 + (1 + 3w)^2$$

This is a quadratic objective function, and the individual losses are the three terms of the above expression. The aggregate loss can also be written as $J = 14w^2 + 3$. Therefore, the loss functions of the three individual points and the aggregate loss are both quadratic. This is the reason that the Newton method converges to the optimal solution in a single step in least-squares classification/regression; the Taylor "approximation" is exact.

Let us now examine, how this objective function would be modified by the L_2-SVM:

$$J = \max\{(1-w), 0\}^2 + \max\{(1-2w), 0\}^2 + \max\{(1+3w), 0\}^2$$

This objective function is no longer quadratic because of the use of the maximization function within the loss. As a result, the Taylor approximation is no longer exact, and a finite step will lead to a point where the Taylor approximation deteriorates. Note that different points contribute non-zero values at different values of w. Therefore, for any Newton step of finite size, points may drop off or add into the loss, which can cause unexpected results. For example, as one reaches near an optimal solution many misclassified training points may be the result of noise and errors in the training data. In this situation, the Newton method will define the update of the weight vector based on such unreliable training points. This is one of the reasons that line search in important in the Newton method. Another solution is to use the *trust region method*.

5.6.3.1 Trust Region Method

The trust-region method can be viewed as a complementary approach to line-search; whereas line-search selects the step-size after choosing the direction, a trust-region method selects the direction after choosing a step-size (trust region), which is incorporated within the optimization formulation for selecting the direction of movement. Let $\overline{W} = \overline{a}_t$ be the value of the parameter vector at the tth iteration of optimizing the objective function $J(\overline{W})$. Similarly, let H_t be the Hessian of the loss function, when evaluated at $\overline{a}_t$. Then, the trust-region method solves the following subproblem using an important quantity $\delta_t > 0$ that controls the trust-region size:

$$\text{Minimize } F(\overline{W}) = J(\overline{a}_t) + (\overline{W} - \overline{a}_t)^T [\nabla J(\overline{a}_t)] + \frac{1}{2}(\overline{W} - \overline{a}_t)^T H_t (\overline{W} - \overline{a}_t)$$

$$\text{subject to:}$$

$$\|\overline{W} - \overline{a}_t\| \leq \delta_t$$

The objective function $F(\overline{W})$ contains the second-order Taylor approximation of the true objective function $J(\overline{W})$ in the locality of the current parameter vector $\overline{a}_t$. Note that this approach is also working with the approximate quadratic bowl like the Newton method, except that it does not move to the bottom of the quadratic bowl. Rather, one uses the trust radius δ_t to restrict the amount of movement as a constraint. This type of restriction is referred to as the *trust constraint*. The key point here is that the direction of best movement is also affected by regulating the maximum step-size, which makes it complementary to line-search methods. For example, if the maximum step-size δ_t is chosen to be very small, then the direction of movement will be very similar to a vanilla gradient-descent method, rather than the inverse-Hessian biased Newton method. The basic idea is that the Taylor approximation becomes less and less reliable with increasing distance from the point of expansion, and therefore one needs to restrict the radius in order to obtain better improvements. The broad process of solving such convex optimization problems with constraints is provided in Chapter 6, and a specific method for solving this type of optimization problem is provided in Section 6.5.1.

A key point is in terms of how the radius δ_t should be selected. The radius δ_t is either increased or decreased, by comparing the improvement $F(\overline{a}_t) - F(\overline{a}_{t+1})$ of the Taylor approximation $F(\overline{W})$ to the improvement $J(\overline{a}_t) - J(\overline{a}_{t+1})$ of the true objective function:

$$I_t = \frac{J(\overline{a}_t) - J(\overline{a}_{t+1})}{F(\overline{a}_t) - F(\overline{a}_{t+1})} \quad \text{[Improvement Ratio]}$$

Intuitively, we would like the true objective function to improve as much as possible, and not just the Taylor approximation. The value of the improvement ratio I_t is usually less than 1, as one is optimizing the Taylor approximation rather than the true objective function. For example, choosing extremely small values of δ_t will lead to improvement ratios near 1, but it is not helpful in terms of making sufficient progress.

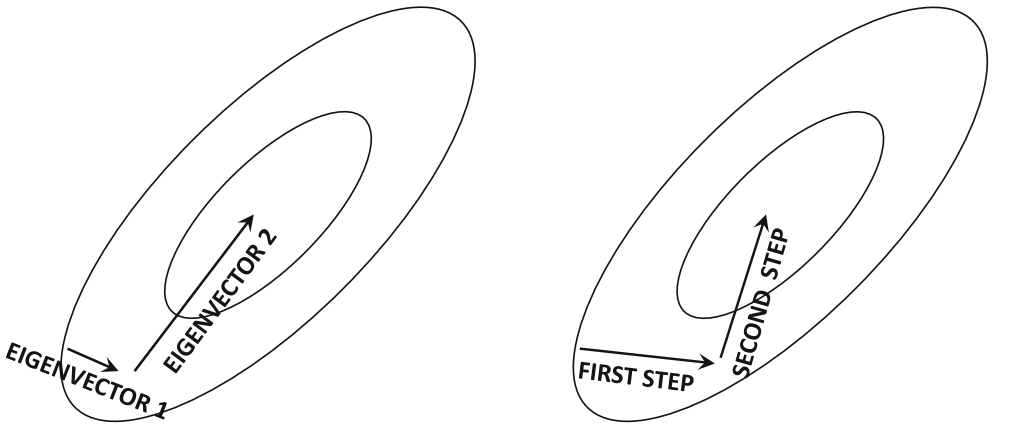
Therefore, the change in δ_t from iteration to iteration is accomplished by using the improvement ratio as a hint about whether it is too conservative or too liberal. Similarly, the trust constraint $\|\overline{W} - \overline{a}_t\| \leq \delta_t$ needs to be satisfied tightly by the optimization solution $\overline{W} = \overline{a}_{t+1}$ in order to increase the size of the trust region in the next iteration. If the improvement ratio is too small (say, less than 0.25), then the trust radius δ_t needs to be reduced by a factor of 2 in the next iteration. If the ratio is too large (say, greater than 0.75) and a full step of δ_t was used in the current iteration (i.e., tightly satisfied trust constraint), the trust radius δ_t needs to be increased. Otherwise, the trust radius does not change. Furthermore, if the improvement ratio is smaller than a critical point (say, negative), then the current step is not accepted, and we set $\overline{a}_{t+1} = \overline{a}_t$ and the optimization problem is solved again with a smaller step size. This process is repeated to convergence. An example of the implementation of logistic regression with a trust-region method is given in [80].

5.7 Computationally Efficient Variations of Newton Method

The Newton method requires fewer iterations than vanilla gradient descent, but each iteration is more expensive. The main challenge arises in the inversion of the Hessian. When the number of parameters is large, the Hessian is too large to store or compute explicitly. This situation arises commonly in domains such as neural network optimization. It is not uncommon to have neural networks with millions of parameters. Trying to compute the inverse of a $10^6 \times 10^6$ Hessian matrix is impractical. Therefore, many approximations and variations of the Newton method have been developed. All these methods borrow the quadratic-approximation principles of the Newton method, but are able to implement these methods more efficiently. Examples of such methods include the method of *conjugate gradients* [19, 59, 86, 87] and quasi-Newton methods that approximate the Hessian. The method of conjugate gradients does not materialize even an approximation of the Hessian, but it tries to express the Newton step as a sequence of d simpler steps, where d is the dimensionality of the data. The d directions of these steps are referred to as *conjugate directions*, which is how this method derives its name. Since the Hessian is never explicitly computed, this technique is also referred to as *Hessian-free optimization*.

5.7.1 Conjugate Gradient Method

The *conjugate gradient method* [59] requires d steps to reach the optimal solution of a quadratic loss function (instead of a single Newton step). The basic idea is that any quadratic function can be transformed to a sum of additively separable univariate functions by using an



(a) Eigenvectors of Hessian Mutually Orthogonal: $\overline{q}_i^T \overline{q}_j = 0$

(b) Arbitrary conjugate pair Non-orthogonal: $\overline{q}_i^T H \overline{q}_j = 0$

Figure 5.10: The eigenvectors of the Hessian of a quadratic function represent the orthogonal axes of the quadratic ellipsoid and are also mutually orthogonal. The eigenvectors of the Hessian are orthogonal conjugate directions. The generalized definition of conjugacy may result in non-orthogonal directions

appropriate basis transformation of variables (cf. Section 3.4.4 of Chapter 3). These variables represent directions in the data that do not interact with one another. Such noninteracting directions are extremely convenient for optimization because they can be independently optimized with line search. Since it is possible to find such directions only for quadratic loss functions, we will first discuss the conjugate gradient method under the assumption that the objective function $J(\overline{W})$ is quadratic. Later, we will discuss the generalization to non-quadratic functions.

A quadratic and convex loss function $J(\overline{W})$ has an ellipsoidal contour plot of the type shown in Figure 5.10, and has a constant Hessian over all regions of the optimization space. The orthonormal eigenvectors $\overline{q}_0 \ldots \overline{q}_{d-1}$ of the symmetric Hessian represent the axes directions of the ellipsoidal contour plot. One can rewrite the loss function in a new coordinate space defined by the eigenvectors as the basis vectors (cf. Section 3.4.4 of Chapter 3) *to create an additively separable sum of univariate quadratic functions in the different variables*. This is because the new coordinate system creates a basis-aligned ellipse, which does not have interacting quadratic terms of the type $x_i x_j$. Therefore, each transformed variable can be optimized independently of the others. Alternatively, one can work with the original variables (without transformation), and simply perform line search along each eigenvector of the Hessian to select the step size. The nature of the movement is illustrated in Figure 5.10(a). Note that movement along the jth eigenvector does not disturb the work done along other eigenvectors, and therefore d steps are sufficient to reach the optimal solution in quadratic loss functions.

Although it is impractical to compute the eigenvectors of the Hessian, there are other efficiently computable directions satisfying similar properties; this key property is referred to as *mutual conjugacy* of vectors. Note that two eigenvectors $\overline{q}_i$ and $\overline{q}_j$ of the Hessian satisfy $\overline{q}_i^T \overline{q}_j = 0$ because of orthogonality of the eigenvectors of a symmetric matrix. Furthermore, since $\overline{q}_j$ is an eigenvector of H, we have $H\overline{q}_j = \lambda_j \overline{q}_j$ for some scalar eigenvalue λ_j. Multiplying both sides with $\overline{q}_i^T$, we can easily show that the eigenvectors of the Hessian satisfy

$\overline{q}_i^T H \overline{q}_j = 0$ in pairwise fashion. The condition $\overline{q}_i^T H \overline{q}_j = 0$ is referred to as H-orthogonality in linear algebra, and is also referred to as the *mutual conjugacy condition* in optimization. It is this mutual conjugacy condition that results in linearly separated variables. However, the eigenvectors are not the only set of mutually conjugate conditions. Just as there are an infinite number of orthonormal basis sets, there are an infinite number of H-orthogonal basis sets in d-dimensional space. In fact, the expression $\langle \overline{q}_i, \overline{q}_j \rangle = \overline{q}_i^T H \overline{q}_j$ is a generalized form of the dot product, referred to as the *inner product*, which has particular significance to quadratic optimization with an elliptical Hessian. If we re-write the quadratic loss function in terms of coordinates in *any* axis system of H-orthogonal directions, the objective function will contain a sum of univariate quadratic functions in terms of the transformed variables. In order to understand why this is the case, let us construct the $d \times d$ matrix $Q = [\overline{q}_0 \ldots \overline{q}_{d-1}]$, whose columns contain H-orthogonal directions. Therefore $\Delta = Q^T H Q$ is diagonal by definition of H-orthogonality. Now note that a quadratic objective function with Hessian H is always of the form $J(\overline{W}) = \overline{W}^T H \overline{W}/2 + \overline{b}^T \overline{W} + c$. Here, $\overline{b}$ is a d-dimensional vector and c is a scalar. This same quadratic function can be expressed in terms of the transformed variables $\overline{W}'$ satisfying $\overline{W} = Q\overline{W}'$ as follows:

$$\begin{aligned} J\left(Q\overline{W}'\right) &= \overline{W}'^T \left[Q^T H Q\right] \overline{W}'/2 + \overline{b}^T Q \overline{W}' + c \\ &= \overline{W}'^T \Delta \overline{W}'/2 + \overline{b}^T Q \overline{W}' + c \end{aligned}$$

Note that the second-order term in the above objective function uses the diagonal matrix Δ, where $\overline{W}'$ contains the coordinates of the parameter vector in the basis corresponding to the conjugate directions. Of course, we do not need to be explicit about performing a basis transformation into an additively separable objective function. Rather, one can separately optimize along each of these d H-orthogonal directions (in terms of the original variables) to solve the quadratic optimization problem in d steps. Each of these optimization steps can be performed using line search along an H-orthogonal direction. Hessian eigenvectors represent a rather special set of H-orthogonal directions that are also orthogonal; conjugate directions other than Hessian eigenvectors, such as those shown in Figure 5.10(b), are not mutually orthogonal. Therefore, conjugate gradient descent optimizes a quadratic objective function by *implicitly* transforming the loss function into a *non-orthogonal* basis with a additively separable representation of the objective function in which each additive term is a univariate quadratic. One can state this observation as follows:

Observation 5.7.1 (Properties of H-Orthogonal Directions) *Let H be the Hessian of a quadratic objective function. If any set of d H-orthogonal directions are selected for movement, then one is implicitly moving along separable variables in a transformed representation of the function. Therefore, at most d steps are required for quadratic optimization.*

The independent optimization along each non-interacting direction (with line search) ensures that the component of the gradient along each conjugate direction will be 0. Strictly convex loss functions have linearly independent conjugate directions (see Exercise 9). In other words, the final gradient will have zero dot product with d linearly independent directions; this is possible only when the final gradient is the zero vector (see Exercise 10), which implies optimality for a convex function. In fact, one can often reach a near-optimal solution in far fewer than d updates.

How can one identify conjugate directions? The simplest approach is to use generalized Gram-Schmidt orthogonalization on the Hessian of the quadratic function in order to generate H-orthogonal directions (cf. Problem 2.7.1 of Chapter 2 and Exercise 11 of this

chapter). Such an orthogonalization is easy to achieve using arbitrary vectors as starting points. However, this process can still be quite expensive because each direction $\overline{q}_t$ needs to use *all* the previous directions $\overline{q}_0 \ldots \overline{q}_{t-1}$ for iterative generation in the Gram-Schmidt method. Since each direction is a d-dimensional vector, and there are $O(d)$ such directions towards the end of the process, it follows that each step will require $O(d^2)$ time. Is there a way to do this using only the previous direction in order to reduce this time from $O(d^2)$ to $O(d)$? Surprisingly, only the most recent conjugate direction is needed to generate the next direction [99, 114], when steepest descent directions are used for iterative generation. In other words, one should not use Gram-Schmidt orthogonalization with arbitrary vectors, but should use steepest descent directions as the raw vectors to be orthogonalized. This choice makes all the difference in ensuring a more efficient form of orthogonalization. This is not an obvious result (see Exercise 12). The direction $\overline{q}_{t+1}$ is, therefore, defined iteratively as a linear combination of *only* the previous conjugate direction $\overline{q}_t$ and the current steepest descent direction $\nabla J(\overline{W}_{t+1})$ with combination parameter β_t:

$$\overline{q}_{t+1} = -\nabla J(\overline{W}_{t+1}) + \beta_t \overline{q}_t \tag{5.21}$$

Premultiplying both sides with $\overline{q}_t^T H$ and using the conjugacy condition to set the left-hand side to 0, one can solve for β_t:

$$\beta_t = \frac{\overline{q}_t^T H[\nabla J(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \tag{5.22}$$

This leads to an iterative update process, which initializes $\overline{q}_0 = -\nabla J(\overline{W}_0)$, and computes $\overline{q}_{t+1}$ iteratively for $t = 0, 1, 2, \ldots T$:

1. Update $\overline{W}_{t+1} \Leftarrow \overline{W}_t + \alpha_t \overline{q}_t$. Here, the step size α_t is computed using line search to minimize the loss function.
2. Set $\overline{q}_{t+1} = -\nabla J(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H[\nabla J(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \right) \overline{q}_t$. Increment t by 1.

It can be shown [99, 114] that $\overline{q}_{t+1}$ satisfies conjugacy with respect to *all* previous $\overline{q}_i$. A systematic road-map of this proof is provided in Exercise 12.

The conjugate-gradient method is also referred to as Hessian-free optimization. However, the above updates do not *seem* to be Hessian-free, because the matrix H is included in the above updates. However, the underlying computations only need the *projection* of the Hessian along particular directions; we will see that these can be computed indirectly using the method of finite differences without explicitly computing the individual elements of the Hessian. Let $\overline{v}$ be the vector direction for which the projection $H\overline{v}$ needs to be computed. The method of finite differences computes the loss gradient at the current parameter vector $\overline{W}$ and at $\overline{W} + \delta\overline{v}$ for some small value of δ in order to perform the approximation:

$$H\overline{v} \approx \frac{\nabla J(\overline{W} + \delta\overline{v}) - \nabla J(\overline{W})}{\delta} \propto \nabla J(\overline{W} + \delta\overline{v}) - \nabla J(\overline{W}) \tag{5.23}$$

The right-hand side is free of the Hessian. The condition is exact for quadratic functions. Other alternatives for Hessian-free updates are discussed in [19].

So far, we have discussed the simplified case of quadratic loss functions, in which the Hessian is a constant matrix (i.e., independent of the current parameter vector). However, most loss functions in machine learning are not quadratic and, therefore, the Hessian matrix is dependent on the current value of the parameter vector $\overline{W}_t$. This leads to several choices

in terms of how one can create a modified algorithm for non-quadratic functions. Do we first create a quadratic approximation at a point and then solve it for a few iterations with the Hessian (quadratic approximation) fixed at that point, or do we change the Hessian every iteration along with the change in parameter vector? The former is referred to as the *linear conjugate gradient method*, whereas the latter is referred to as the *nonlinear conjugate gradient method*.

In the nonlinear conjugate gradient method, the mutual conjugacy (i.e., H-orthogonality) of the directions will deteriorate over time, as the Hessian changes from one step to the next. This can have an unpredictable effect on the overall progress from one step to the next. Furthermore, the computation of conjugate directions needs to be restarted every few steps, as the mutual conjugacy deteriorates. If the deterioration occurs too fast, the restarts occur very frequently, and one does not gain much from conjugacy. On the other hand, each quadratic approximation in the linear conjugate gradient method can be solved exactly, and will typically be (almost) solved in much fewer than d iterations. Therefore, one can make similar progress to the Newton method in each iteration. As long as the quadratic approximation is of high quality, the required number of approximations is often not too large. The nonlinear conjugate gradient method has been extensively used in traditional machine learning from a historical perspective [19], although recent work [86, 87] has advocated the use of linear conjugate methods. Experimental results in [86, 87] suggest that linear conjugate gradient methods have some advantages.

5.7.2 Quasi-Newton Methods and BFGS

The acronym BFGS stands for the Broyden–Fletcher–Goldfarb–Shanno algorithm, and it is derived as an approximation of the Newton method. Let us revisit the updates of the Newton method. A typical update of the Newton method is as follows:

$$\overline{W}^* \Leftarrow \overline{W}_0 - H^{-1}[\nabla J(\overline{W}_0)] \tag{5.24}$$

In quasi-Newton methods, a sequence of approximations of the inverse Hessian matrix are used in various steps. Let the approximation of the inverse Hessian matrix in the tth step be denoted by G_t. In the very first iteration, the value of G_t is initialized to the identity matrix, which amounts to moving along the steepest-descent direction. This matrix is continuously updated from G_t to G_{t+1} with low-rank updates (derived from the matrix inversion lemma of Chapter 1). A direct restatement of the Newton update in terms of the inverse Hessian $G_t \approx H_t^{-1}$ is as follows:

$$\overline{W}_{t+1} \Leftarrow \overline{W}_t - G_t[\nabla J(\overline{W}_t)] \tag{5.25}$$

The above update can be improved with an optimized learning rate α_t for non-quadratic loss functions working with (inverse) Hessian approximations like G_t:

$$\overline{W}_{t+1} \Leftarrow \overline{W}_t - \alpha_t G_t[\nabla J(\overline{W}_t)] \tag{5.26}$$

The optimized learning rate α_t is identified with line search. The line search does not need to be performed exactly (like the conjugate gradient method), because maintenance of conjugacy is no longer critical. Nevertheless, approximate conjugacy of the early set of directions is maintained by the method when starting with the identity matrix. One can (optionally) reset G_t to the identity matrix every d iterations (although this is rarely done).

It remains to be discussed how the matrix G_{t+1} is approximated from G_t. For this purpose, the *quasi-Newton condition*, also referred to as the *secant condition*, is needed:

$$\underbrace{\overline{W}_{t+1} - \overline{W}_t}_{\text{Parameter Change}} = G_{t+1} \underbrace{[\nabla J(\overline{W}_{t+1}) - \nabla J(\overline{W}_t)]}_{\text{First derivative change}} \tag{5.27}$$

The above formula is simply a finite-difference approximation. Intuitively, multiplication of the second-derivative matrix (i.e., Hessian) with the parameter change (vector) approximately provides the gradient change. Therefore, multiplication of the inverse Hessian approximation G_{t+1} with the gradient change provides the parameter change. The goal is to find a symmetric matrix G_{t+1} satisfying Equation 5.27, but it represents an underdetermined system of equations with an infinite number of solutions. Among these, BFGS chooses the closest symmetric G_{t+1} to the current G_t, and achieves this goal by posing a minimization objective function $\|G_{t+1} - G_t\|_w$ in the form of a *weighted Frobenius norm*. In other words, we want to find G_{t+1} satisfying the following:

$$\begin{aligned} \text{Minimize}_{[G_{t+1}]} \; & \|G_{t+1} - G_t\|_w \\ & \text{subject to:} \\ & \overline{W}_{t+1} - \overline{W}_t = G_{t+1}[\nabla J(\overline{W}_{t+1}) - \nabla J(\overline{W}_t)] \\ & G_{t+1}^T = G_{t+1} \end{aligned}$$

The subscript of the norm is annotated by "w" to indicate that it is a weighted[3] form of the norm. This weight is an "averaged" form of the Hessian, and we refer the reader to [99] for details of how the averaging is done. Note that one is not constrained to using the weighted Frobenius norm, and different variations of how the norm is constructed lead to different variations of the quasi-Newton method. For example, one can pose the same objective function and secant condition in terms of the Hessian rather than the inverse Hessian, and the resulting method is referred to as the Davidson–Fletcher–Powell (DFP) method. In the following, we will stick to the use of the inverse Hessian, which is the BFGS method.

Since the weighted norm uses the Frobenius matrix norm (along with a weight matrix) the above is a quadratic optimization problem with linear constraints. Such constrained optimization problems are discussed in Chapter 6. In general, when there are linear equality constraints paired with a quadratic objective function, the structure of the optimization problem is quite simple, and closed-form solutions can sometimes be found. This is because the equality constraints can often be eliminated along with corresponding variables (using methods like Gaussian elimination), and an unconstrained, quadratic optimization problem can be defined in terms of the remaining variables. These problems sometimes turn out to have closed-form solutions like least-squared regression. In this case, the closed-form solution to the above optimization problem is as follows:

$$G_{t+1} \Leftarrow (I - \Delta_t \overline{q}_t \overline{v}_t^T) G_t (I - \Delta_t \overline{v}_t \overline{q}_t^T) + \Delta_t \overline{q}_t \overline{q}_t^T \tag{5.28}$$

Here, the (column) vectors $\overline{q}_t$ and $\overline{v}_t$ represent the parameter change and the gradient change; the scalar $\Delta_t = 1/(\overline{q}_t^T \overline{v}_t)$ is the inverse of the dot product of these two vectors.

$$\overline{q}_t = \overline{W}_{t+1} - \overline{W}_t; \quad \overline{v}_t = \nabla L(\overline{W}_{t+1}) - \nabla L(\overline{W}_t)$$

[3]The form of the objective function is $\|A^{1/2}(G_{t+1} - G_t)A^{1/2}\|_F$ norm, where A is an averaged version of the Hessian matrix over various lengths of the step. We refer the reader to [99] for details.

The update in Equation 5.28 can be made more space efficient by expanding it, so that fewer temporary matrices need to be maintained. Interested readers are referred to [83, 99, 104] for implementation details and derivation of these updates.

Even though BFGS benefits from approximating the inverse Hessian, it does need to carry over a matrix G_t of size $O(d^2)$ from one iteration to the next. The *limited memory BFGS* (L-BFGS) reduces the memory requirement drastically from $O(d^2)$ to $O(d)$ by not carrying over the matrix G_t from the previous iteration. In the most basic version of the L-BFGS method, the matrix G_t is replaced with the identity matrix in Equation 5.28 in order to derive G_{t+1}. A more refined choice is to store the $m \approx 30$ most recent vectors $\overline{q}_t$ and $\overline{v}_t$. Then, L-BFGS is equivalent to initializing G_{t-m+1} to the identity matrix and recursively applying Equation 5.28 m times to derive G_{t+1}. In practice, the implementation is optimized to directly compute the direction of movement from the vectors without explicitly storing large intermediate matrices from G_{t-m+1} to G_t.

5.8 Non-differentiable Optimization Functions

Several optimization functions in machine learning are non-differentiable. A mild example is the case in which an L_1-loss or L_1-regularization is used. A key point is that any type of L_1-norm of the vector $\overline{v} = [v_1, \ldots, v_d]$ uses the modulus $|v_i|$ of each of the vector components in the norm $\sum_{i=1}^{d} |v_i|$. The derivative of $|v_i|$ is non-differentiable at $v_i = 0$. Furthermore, any type of L_1-loss is non-differentiable. For example, the hinge loss of the support vector machine is non-differentiable.

A more severe form of non-differentiability is one in which one is trying optimize an inherently discrete objective function such as a *ranking objective function*. In many rare-class settings of classification, one of the labels is far less frequent compared to the others. For example, in a labeled database of intrusion records, the intrusion records are likely to less frequent compared to the normal records. In such cases, the objective function is often defined based on a function of the ranking of *instances with respect to their propensity to belong to the rare class*. For example, one might minimize the sum of (algorithm-determined) ranks of instances that truly belong to the rare class (based on ground-truth information). Note that this is a non-differentiable function because significant changes in the parameter vector might sometimes not affect the algorithmic ranking at all, and at other times infinitesimal changes in parameters might drastically affect the ranking. This results in a loss function with vertical walls and flat regions. As a specific example, consider a 1-dimensional example, in which the points are ranked according to decreasing value of $w \cdot x$, where x is the 1-dimensional feature value and w is the scalar parameter. The four training-label pairs are $(1, +1)$, $(2, +1)$, $(-1, -1)$, and $(-2, -1)$. Ideally, we would like to choose values of w so that all positive examples are ranked above the negative examples. In this simple problem, choosing any value $w > 0$ provides an ideal ranking in which the two positive examples have ranks of 1 and 2. Therefore, the sum of the ranks of positive instances is 3. Choosing $w < 0$ provides the worst-possible ranking in which the two positive instances have ranks of 3 and 4 (with a sum of 7). Choosing $w = 0$ leads to a tied rank of 2.5 for all training instances, and the sum of the ranks is 5. The objective function corresponding to the sum of the ranks (of only the positive instances) is shown in Figure 5.11. The problem with this staircase-like objective function is that it is not really informative anywhere from the perspective of gradient descent. Although the loss function is differentiable almost everywhere except for a single point, the zero gradient at all points provides no clues about the best direction of descent.

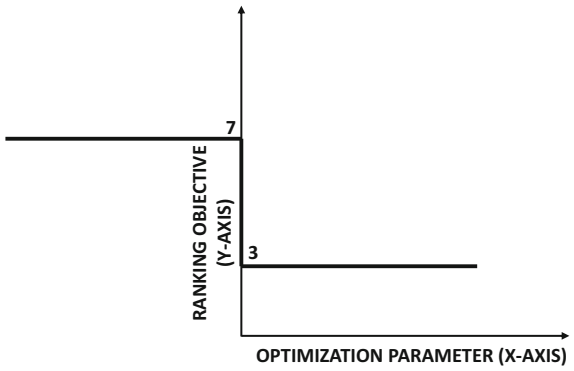


Figure 5.11: An example of a non-differentiable optimization problem caused by a ranking objective function

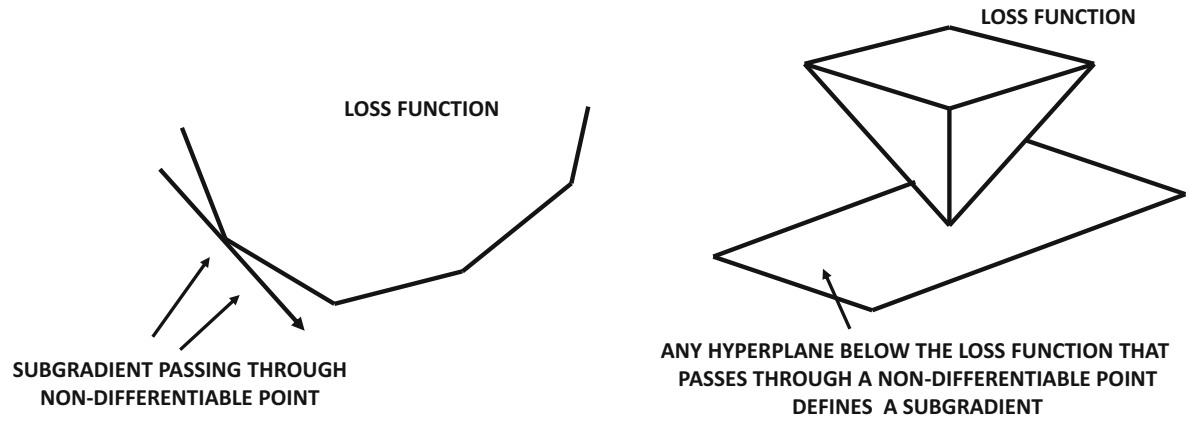
These types of non-differentiability are often addressed by either making fundamental changes to the underlying optimization algorithms, or by changing the loss function in order to make it smooth. After all, the loss functions of machine learning algorithms are almost always smooth approximations to discrete objective functions (like classification accuracy). In the following, we will provide an overview of the different types of methods used to handle non-differentiability in machine learning.

5.8.1 The Subgradient Method

The subgradient method is designed to work for convex minimization problems, where the gradient is informative at most points except for a few specific points where the objective function is non-differentiable. In such cases, subgradient mainly serves the purpose of bringing the optimization problem out of its non-differentiable "rut." Since the function is differentiable at most other points, it does not face many challenges in terms of optimization, once it gets out of this non-differentiable rut.

The main issue with non-differentiability is that the *one-sided derivatives* are different. For example, $|x|$ has a right-derivative of $+1$ and a left-derivative of -1. A subgradient corresponds to the interval $[-1, +1]$. The presence of the zero vector among the subgradients is an optimality condition for the subgradient method. In Figure 5.12(a), one possible subgradient of a 1-dimensional function is illustrated. Intuitively, the subgradient always lies "below" the loss function, as shown in Figure 5.12(a). Note that there are many possible subgradients in this case because one can construct the line below the loss function in many possible ways. For the d-dimensional function corresponding to the L_1-norm $\|\overline{w}\|_1$ of $\overline{w}$, one can select any d-dimensional vector for which each component is sampled uniformly at random from $(-1, 1)$ to create a subgradient. In Figure 5.12(a), we have shown an example of a subgradient for a 1-dimensional function. Note that one can draw many possible "tangents" at non-differentiable points for convex functions, which are (more precisely) referred to as *subtangents* at non-differentiable points. Each of these subtangents corresponds to a subgradient. For multidimensional functions, the subgradient is defined by any hyperplane lying fully below the loss function, as shown in Figure 5.12(b). For differentiable functions, we can draw only one tangent hyperplane. However, non-differentiable functions allow the construction of an infinite number of possibilities.

A subgradient of a function $J(\overline{w})$ at point $\overline{w}_0$ is formally defined as follows:



(a) Subgradient in one dimension

(b) Subgradient in two dimensions

Figure 5.12: Subgradients in one and two dimensions. Any vector residing on the hyperplane, which originates at the contact point between the loss function and the hyperplane, is a subgradient. The vertical direction is the loss function value in each case

Definition 5.8.1 (Subgradient) *Let $J(\overline{w})$ be a multivariate, convex loss function in d dimensions. The subgradient at point $\overline{w}_0$ is a d-dimensional vector $\overline{v}$ that satisfies the following for any $\overline{w}$:*

$$J(\overline{w}) \geq J(\overline{w}_0) + \overline{v} \cdot (\overline{w} - \overline{w}_0)$$

Note that the notion of subgradient is primarily used in a *convex* function rather than an arbitrary function (as in conventional gradients). Although it is possible to also apply the above definition for nonconvex functions, the definition loses its usefulness in those cases. The subgradient is not unique unless the function is differentiable at that point. At differentiable points, the subgradient is simply the gradient. It can be shown that any convex combination of subgradients is a subgradient.

Problem 5.8.1 *Show using Definition 5.8.1 that if $\overline{v}_1$ and $\overline{v}_2$ are subgradients of $J(\overline{w})$ at $\overline{w} = \overline{w}_0$, then $\lambda\overline{v}_1 + (1-\lambda)\overline{v}_2$ is also a subgradient of $J(\overline{w})$ for any $\lambda \in (0, 1)$.*

The above practice problem shows that the set of subgradients is a convex closed set. Furthermore, if the zero vector is a subgradient at $\overline{w}_0$, then Definition 5.8.1 implies that we have $J(\overline{w}) \geq J(\overline{w}_0)$ for all $\overline{w}$. In other words, $\overline{w}_0$ is an optimal solution. In the following, we mention some key properties of subgradients:

1. The conventional gradient at a differentiable point is its unique subgradient.

2. For convex functions, the optimality condition for a particular value of the optimization variables $\overline{w}_0$ is that the set of subgradients at $\overline{w}_0$ must include the zero vector.

3. At any point $\overline{w}_0$, the sum of any subgradient of $J_1(\overline{w}_0)$ and any subgradient of $J_2(\overline{w}_0)$ is a subgradient of $(J_1 + J_2)(\overline{w}_0)$. In other words, we can decompose the subgradient of a separably additive function into its constituent subgradients. This property is relevant to loss functions of various machine learning algorithms that add up loss contributions of individual training points.

While it might not be immediately obvious, we have already used the subgradient method (implicitly) in the hinge-loss SVM in Chapter 4. We repeat the objective function of the hinge-loss SVM here (cf. page 184), which is based on the training pairs $(\overline{X}_i, y_i)$:

$$J = \sum_{i=1}^{n} \max\{0, (1 - y_i[\overline{W}^T \cdot \overline{X}_i])\} + \frac{\lambda}{2}\|\overline{W}\|^2 \quad \text{[Hinge-loss SVM]}$$

As evident from Figure 4.9 of Chapter 4, the use of the maximization function causes non-differentiability at the sharp "hinge" of the hinge-loss function; these are values of $\overline{W}$ where the second argument of the max-function is 0 for any training point. So what happens at these points? The update of the SVM uses only those training points where the second argument is *not* zero. Therefore, at the non-differentiable points, the gradient is simply set to 0, which is a valid subgradient. Therefore, the primal updates of the hinge-loss SVM implicitly use the subgradient method, although the use is straightforward and natural. In this case, the subgradient does not point in a direction of instantaneous movement that *worsens* the objective function (for infinitesimal steps). This is not the case for more aggressive uses of the subgradient method.

5.8.1.1 Application: L_1-Regularization

A more aggressive use of the subgradient method appears in least-squares regression with L_1-regularization.

$$\text{Minimize } J = \underbrace{\frac{1}{2}\|D\overline{W} - \overline{y}\|^2}_{\text{Prediction Error}} + \underbrace{\lambda \sum_{j=1}^{d} |w_j|}_{L_1\text{-Regularization}}$$

Here D is an $n \times d$ data matrix whose rows contain the training instances, and $\overline{y}$ is an n-dimensional column vector containing the target variables. The column vector $\overline{W}$ contains the coefficients. Note that the regularization term now uses the L_1-norm of the coefficient vector rather than the L_2-norm. The function J is non-differentiable for any $\overline{W}$ in which even a single component w_j is 0. Specifically, if w_j is infinitesimally larger than 0, then the partial derivative of $|w_j|$ is $+1$, whereas if w_j is infinitesimally smaller than 0, then the partial derivative of $|w_j|$ is -1. In these methods, the partial derivative of w_j at 0 is selected randomly from $[-1, +1]$, whereas the derivative at values different from 0 is computed in the same way as the gradient. Let the subgradient of w_j be denoted by s_j. Then, for step-size $\alpha > 0$, the update is as follows:

$$\overline{W} \Leftarrow \overline{W} - \alpha\,\lambda\,[s_1, s_2, \ldots, s_d]^T - \alpha D^T \underbrace{(D\overline{W} - \overline{y})}_{\text{Error}}$$

Here, each s_j is the subgradient of w_j and is defined as follows:

$$s_j = \begin{cases} -1 & w_j < 0 \\ +1 & w_j > 0 \\ \text{Sample from } [-1, +1] & w_j = 0 \end{cases} \tag{5.29}$$

In this particular case, movement along the subgradient might worsen the objective function value because of the random choice of s_j from $[-1, +1]$. Therefore, one always maintains

the best possible value of $\overline{W}_{best}$ that was obtained in any iteration. At the beginning of the process, both $\overline{W}$ and $\overline{W}_{best}$ are initialized to the same random vector. After each update of $\overline{W}$, the objective function value is evaluated with respect to $\overline{W}$, and $\overline{W}_{best}$, and is set to the recently updated $\overline{W}$ if the objective function value provided by $\overline{W}$ is better than that obtained by the stored value of $\overline{W}_{best}$. At the end of the process, the vector $\overline{W}_{best}$ is returned by the algorithm as the final solution. Note that $s_j = 0$ is also a subgradient at $w_j = 0$, and it is a choice that is sometimes used.

5.8.1.2 Combining Subgradients with Coordinate Descent

The subgradient method can also be combined with coordinate descent (cf. Section 4.10 of Chapter 4) by applying the subgradient optimality condition to the coordinate being learned. The learning problem is often greatly simplified in coordinate descent because only one variable is optimized at a time. As in all coordinate descent methods, one cycles through all the variables one by one in order to perform the optimization.

We provide an example of the use of coordinate descent in linear regression. As in the previous section, let D be an $n \times d$ data matrix with rows containing training instances, and $\overline{y}$ be an n-dimensional column vector of response variables. The d-dimensional column vector of parameters is denoted by $\overline{W} = [w_1 \ldots w_d]^T$. The objective function of least-squares regression with L_1-regularization is repeated below:

$$\text{Minimize } J = \underbrace{\frac{1}{2}\|D\overline{W} - \overline{y}\|^2}_{\text{Prediction Error}} + \underbrace{\lambda \sum_{j=1}^{d} |w_j|}_{L_1\text{-Regularization}}$$

As discussed in Section 4.10 of Chapter 4, coordinate descent can sometimes get stuck for non-differentiable functions. However, a sufficient condition for coordinate descent to work for convex loss functions is that the non-differentiable portion can be decomposed into separable univariate functions (cf. Lemma 4.10.1 of Chapter 4). In this case, the regularization term is clearly a sum of separable and convex functions. Therefore, one can use coordinate descent without getting stuck at a local optimum. The subgradient with respect to *all* the variables is as follows:

$$\nabla J = D^T(D\overline{W} - \overline{y}) + \lambda[s_1, s_2, \ldots s_d]^T \tag{5.30}$$

Here, each s_i is a subgradient drawn from $[-1, +1]$. Since we are optimizing with respect to only the ith variable, we only need to set the ith component of ∇J to zero. Let $\overline{d}_i$ be the ith column of D. Furthermore, let $\overline{r}$ denote the n-dimensional residual vector $\overline{y} - D\overline{W}$. One can then write the optimality condition for the ith component in terms of these variables as follows:

$$\begin{aligned}
&\overline{d}_i^T(\overline{y} - D\overline{W}) - \lambda s_j = 0 \\
&\overline{d}_i^T \overline{r} - \lambda s_j = 0 \\
&\overline{d}_i^T \overline{r} + w_i \overline{d}_i^T \overline{d}_i - \lambda s_j = w_i \overline{d}_i^T \overline{d}_i
\end{aligned}$$

The left-hand side is free of w_i because the term $\overline{d}_i^T \overline{r}$ contributes $-w_i \overline{d}_i^T \overline{d}_i$, which cancels with $w_i \overline{d}_i^T \overline{d}_i$. Therefore, we obtain the coordinate update for w_i:

$$\overline{w}_i \Leftarrow \overline{w}_i + \frac{\overline{d}_i^T \overline{r} - \lambda s_i}{\|\overline{d}_i\|^2} \tag{5.31}$$

The value of the subgradient s_i is defined in the same way as in the previous section. The main problem is that each s_i could be chosen to be any value between -1 and $+1$ when the updated value of w_i is close enough to 0; only one of these values will arrive at the optimal solution. How can one determine the exact value of s_i that optimizes the objective function in such cases? This is achieved by the use of *soft thresholding* of such "close enough" values of w_i to 0. Soft thresholding of w_i automatically sets the value of s_i to an appropriate intermediate value between -1 and $+1$. Therefore, the value of each w_i is set as follows:

$$w_i \Leftarrow \begin{cases} 0, & -\frac{\lambda}{\|\overline{d}_i\|^2} \leq \overline{w}_i + \frac{\overline{d}_i^T \overline{r}}{\|\overline{d}_i\|^2} \leq \frac{\lambda}{\|\overline{d}_i\|^2} \\ \overline{w}_i + \frac{\overline{d}_i^T \overline{r} - \lambda \text{sign}(w_i)}{\|\overline{d}_i\|^2}, & \text{otherwise} \end{cases} \tag{5.32}$$

As in any form of coordinate-descent, one cycles through the variables one by one until convergence is reached. The *elastic-net* combines both L_1- and L_2-regularization, and we leave the derivation of the resulting updates as a practice problem.

Problem 5.8.2 (Elastic-Net Regression) *Consider the problem of elastic-net regression with the following objective function:*

$$\textit{Minimize } J = \frac{1}{2}\|D\overline{W} - \overline{y}\|^2 + \lambda_1 \sum_{j=1}^{d} |w_j| + \frac{\lambda_2}{2} \sum_{j=1}^{d} w_j^2$$

Show that the updates of coordinate decent can be expressed as follows:

$$w_i \Leftarrow \begin{cases} 0, & -\frac{\lambda_1}{\|\overline{d}_i\|^2+\lambda_2} \leq \frac{w_i\|\overline{d}_i\|^2 + \overline{d}_i^T \overline{r}}{\|\overline{d}_i\|^2+\lambda_2} \leq \frac{\lambda_1}{\|\overline{d}_i\|^2+\lambda_2} \\ \frac{w_i\|\overline{d}_i\|^2 + \overline{d}_i^T \overline{r} - \lambda_1 sign(w_i)}{\|\overline{d}_i\|^2+\lambda_2}, & otherwise \end{cases}$$

The main challenge in coordinate descent is to avoid getting stuck in a local optimum because of non-differentiability (see Figure 4.10 of Chapter 4 for an example). In many cases, one can use variable transformations to convert the objective function to a well-behaved form (cf. Lemma 4.10.1) in which convergence to a global optimum is guaranteed. An example is the *graphical lasso* [48], which implicitly uses variable transformations.

5.8.2 Proximal Gradient Method

The proximal gradient method is particularly useful when the optimization function $J(\overline{W})$ can be broken up into two parts $G(\overline{W})$ and $H(\overline{W})$, one of which is differentiable, and the other is not:

$$J(\overline{W}) = G(\overline{W}) + H(\overline{W})$$

In this form, the portion $G(\overline{W})$ is assumed to be differentiable, whereas $H(\overline{W})$ is not. Both functions are assumed to be convex. The proximal gradient method uses an iterative approach, in which each iteration taking a *gradient step* on $G(\cdot)$ and a *proximal step* on $H(\cdot)$. The proximal step is essentially a minimum value of $H(\cdot)$ in the *locality* of the current value of the parameter vector $\overline{W} = \overline{w}$. This type of minimum in a local region around $\overline{w}$ may be discovered by adding a quadratic penalty to $H(\overline{w})$ depending on how far one ventures from the current value of the parameter vector. Here, a key point is to define the *proximal operator* for the function $H(\cdot)$. The proximal operator $\mathcal{P}$ is defined with the use of a step-size parameter α as follows:

$$\mathcal{P}_{H,\alpha}(\overline{w}) = \text{argmin}_{\overline{u}} \left\{ \alpha H(\overline{u}) + \frac{1}{2}\|\overline{u} - \overline{w}\|^2 \right\} \tag{5.33}$$

In other words, we are trying to minimize the function $H(\cdot)$ in the proximity of $\overline{w}$ by adding a quadratic penalty term to penalize distance from $\overline{w}$. Therefore, the proximity operator will try to find a "better" $\overline{u}$ than $\overline{w}$, but only in the proximity of $\overline{w}$ because distance from $\overline{w}$ is quadratically penalized. Now let us examine what happens with a few examples:

- When $H(\overline{w})$ is set to be a constant, the $\mathcal{P}_{H,\alpha}(\overline{w}) = \overline{w}$. This is because one cannot improve $\overline{w}$ any further from its current argument, and the quadratic penalty encourages staying at the current point.

- When $H(\overline{w})$ is differentiable, then the proximity operator makes an approximate gradient-descent move at step size α. One can derive this result by setting the gradient of the expression inside the argmin of Equation 5.33 to 0:

$$\overline{u} = \overline{w} - \alpha \frac{\partial H(\overline{u})}{\partial \overline{u}} \tag{5.34}$$

  Note that this step is similar to gradient-descent except that the gradient of $H(\cdot)$ is computed at $\overline{u}$ rather than $\overline{w}$. However, the quadratic penalization ensures that the step-size is relatively small, and the computation of the gradient of $H(\overline{u})$ happens only in the proximity of $\overline{w}$. This is a key motivational point. The proximity operator makes sensible moves when $H(\cdot)$ is differentiable. However, it works for non-differentiable functions as well.

Armed with this definition of the proximal operator, one can then write the proximal gradient algorithm in terms of repeating the following two iterative steps as follows:

1. Make a standard gradient-descent step on the differentiable function $G(\cdot)$ with step-size α:

$$\overline{w} \Leftarrow \overline{w} - \alpha \left[\frac{\partial G(\overline{w})}{\partial \overline{w}} \right]$$

2. Make a proximal descent step on the non-differentiable function $H(\cdot)$ with step-size α:

$$\overline{w} \Leftarrow \mathcal{P}_{H,\alpha}(\overline{w}) = \text{argmin}_{\overline{u}} \left\{ \alpha H(\overline{u}) + \frac{1}{2} \|\overline{u} - \overline{w}\|^2 \right\}$$

Note that if the function $H(\cdot)$ is differentiable, then the approach roughly simplifies to alternate gradient descent on $G(\cdot)$ and $H(\cdot)$.

Another key point is in terms of how hard it is to compute the proximal operator. The approach is only used for problems with "simple" proximal operators that are easy to compute; furthermore, the underlying functions have a small number of non-differentiable points. A typical example of such a non-differentiable function is the L_1-norm of a vector. For this reason, the proximal method is less general than the subgradient method; however, when it works, it provides better performance.

5.8.2.1 Application: Alternative for L_1-Regularized Regression

In the previous section, we introduced a subgradient method for least-squares regression with L_1-regularization. In this section, we discuss an alternative based on the proximal gradient method. We rewrite the objective function of least-squares regression and separate it out into the differentiable and non-differentiable parts as follows:

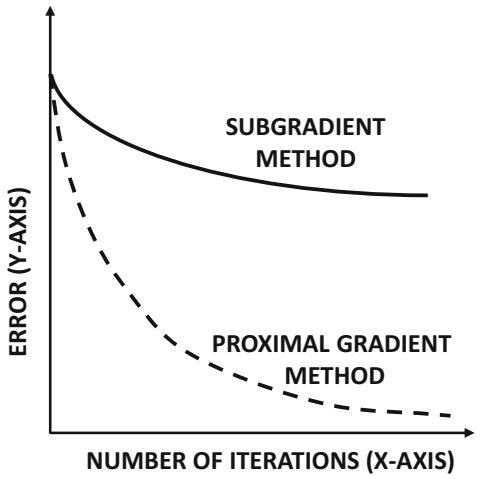


Figure 5.13: An illustrative comparison of the subgradient and the proximal gradient method in terms of typical behavior

$$\text{Minimize } J = \underbrace{\frac{1}{2}\|D\overline{W} - \overline{y}\|^2}_{G(\overline{W})} + \underbrace{\lambda \sum_{j=1}^{d} |w_j|}_{H(\overline{W})}$$

A key point is the definition of the proximal operator on the function $H(\overline{W})$, which the L_1-norm of $\overline{W}$. The proximal operator for $H(\overline{w})$ with step-size α is as follows:

$$[\mathcal{P}_{H,\alpha}]_j = \begin{cases} w_j + \alpha\lambda & w_j < -\alpha\lambda \\ 0 & -\alpha\lambda \leq w_j \leq \alpha\lambda \\ w_j - \alpha\lambda & w_j > \alpha\lambda \end{cases} \tag{5.35}$$

Note that the proximity operator essentially shrinks each w_j by exactly $\alpha\lambda$ as long as it is far away from the non-differentiable point. However, if it is close enough to the non-differentiable point then it simply moves to 0. This is the main difference from the subgradient method, which always updates by exactly $\alpha\lambda$ in either direction at all differentiable points, and updates by a random sample from $[-\alpha\lambda, \alpha\lambda]$ at the non-differentiable point. As a result, the subgradient method is more likely to oscillate around non-differentiable points as compared to the proximal gradient method. An illustrative comparison of the "typical" convergence behavior of the subgradient and proximal gradient method is shown in Figure 5.13. In most cases, the proximal gradient method performance *significantly* faster than the subgradient method. The faster convergence is because of the thresholding approach used in the neighborhood of non-differentiable points. This approach is referred to as the *iterative soft thresholding algorithm*, or *ISTA* in short.

5.8.3 Designing Surrogate Loss Functions for Combinatorial Optimization

Some problems like optimizing the ranking of a set of training instances are inherently combinatorial in nature, which do not provide informative loss surfaces in most regions of the space. For example, as shown in Figure 5.11, the sum of the ranks of positive class instances

results in a highly non-informative function for the purposes of optimization. This function is not only non-differentiable at several points, but its staircase-like nature makes the gradient zero at all differentiable points. In other words, a gradient descent procedure would not know which direction to proceed. This type of problem does not occur with objective functions like the L_1-norm (which enables the use of a subgradient method). In such cases, it makes sense to design a surrogate loss function for the optimization problem at hand. This approach is inherently not a new one; *almost all objective functions for classification are surrogate loss functions anyway.* Strictly speaking, a classification problem should be directly optimizing the classification accuracy with respect to the parameter $\overline{W}$. However, the classification accuracy is another staircase-like function. Therefore, all the models we have seen so far use some form of surrogate loss, such as the least-squares (classification) loss, the hinge loss, and the logistic loss. Extending such methods to ranking problems is therefore not a fundamental innovation at least from a methodological point of view. However, the solutions to ranking objective functions have their own unique characteristics. In the following, we examine some surrogate objective functions designed for the ranking problem for classification.

Most classification objective functions are designed to penalize accuracy of classification by using some surrogate loss, such as the hinge-loss (which is a one-sided penalty from the target values of $+1$ and -1). Ranking-based objective functions are based on exactly the same principle. The only difference is that we penalize the deviation from an ideal ranking with a surrogate loss function. Two examples of such loss functions correspond to the *pairwise* and the *listwise* approaches. In the following, we discuss a simple pairwise approach for defining the loss function.

5.8.3.1 Application: Ranking Support Vector Machine

We will now formalize the optimization model for the ranking SVM. First, the training data is converted into pair-wise examples. For example, in the rare-class ranking problem, one would create pairs of positive and negative class instances, and always rank the positive class above the negative class. The training data $\mathcal{D}_R$ contains the following set of ranked pairs:

$$\mathcal{D}_R = \{(\overline{X}_i, \overline{X}_j) : \overline{X}_i \text{ should be ranked above } \overline{X}_j\}$$

For each such pair in the ranking support vector machine, the goal is learn a d-dimensional weight vector $\overline{W}$, so that $\overline{W} \cdot \overline{X}_i^T > \overline{W} \cdot \overline{X}_j^T$ when $\overline{X}_i$ is ranked above $\overline{X}_j$. Therefore, given an unseen set of test instances $\overline{Z}_1 \ldots \overline{Z}_t$, we can compute each $\overline{W} \cdot \overline{Z}_i^T$, and rank the test instances on the basis of this value.

In the traditional support vector machine, we always impose a *margin requirement* by penalizing points that are uncomfortably close to the decision boundary. Correspondingly, in the ranking SVM, we penalize pairs where the difference between $\overline{W} \cdot \overline{X}_i^T$ and $\overline{W} \cdot \overline{X}_j^T$ is not sufficiently large. Therefore, we would like to impose the following stronger requirement:

$$\overline{W} \cdot (\overline{X}_i - \overline{X}_j)^T > 1$$

Any violations of this condition are penalized by $1 - \overline{W} \cdot (\overline{X}_i - \overline{X}_j)^T$ in the objective function. Therefore, one can formulate the problem as follows:

$$\text{Minimize } J = \sum_{(\overline{X}_i, \overline{X}_j) \in \mathcal{D}_R} \max\{0, [1 - (\overline{W} \cdot [\overline{X}_i - \overline{X}_j]^T)]\} + \frac{\lambda}{2} \|\overline{W}\|^2$$

Here, $\lambda > 0$ is the regularization parameter. Note that one can replace each pair $(\overline{X}_i, \overline{X}_j)$ with the new set of features $\overline{X}_i - \overline{X}_j$. In other words, each $\overline{U}_p$ is of the form $\overline{U}_p = \overline{X}_i - \overline{X}_j$ for a ranked pair $(\overline{X}_i, \overline{X}_j)$ in the training data. Then, the ranking SVM formulates the following optimization problem for the t different pairs in the training data with corresponding features $\overline{U}_1 \ldots \overline{U}_t$:

$$\text{Minimize } J = \sum_{i=1}^{t} \max\{0, [1 - \overline{W} \cdot \overline{U}_i^T]\} + \frac{\lambda}{2}\|\overline{W}\|^2$$

Note that the only difference from a traditional support-vector machine is that the class variable y_i is missing in this optimization formulation. However, this change is extremely easy to incorporate in all the optimization techniques discussed in Section 4.8.2 of Chapter 4. In each case, the class variable y_i is replaced by 1 in the corresponding gradient-descent steps of various methods discussed in Section 4.8.2.

5.8.4 Dynamic Programming for Optimizing Sequential Decisions

Dynamic programming is an approach that is used for optimizing sequential decisions, and the most well-known machine learning application of this approach occurs in *reinforcement learning* [6]. The most general form of reinforcement learning optimizes an objective function $J(a_1 \ldots a_m)$, where $a_1 \ldots a_m$ is a sequence of actions or decisions. For example, finding a shortest path or a longest path from one point to another in a directed acyclic graph requires a sequence of decisions as to which node to select in the next step. Similarly, a two-player game like tic-tac-toe also requires a sequence of decisions about moves to be made in the game, although alternate decisions are made by opponents, and have opposite goals. This principle is used for game learning strategies in reinforcement learning. Another example is that of finding the edit distance between two strings, which requires a sequence of decisions of which edits to make. In all these cases, one has a sequence of decisions $a_1 \ldots a_m$ to make, and after making a decision, one is left with a smaller subproblem to solve. For example, if one has to choose the shortest path from source to sink in a graph, then after choosing the first outgoing node i from the source, one still has to compute the shortest path from i to the sink. In other words, dynamic programming breaks up a larger problems into smaller problems, *each of which would need to be optimally solved*. Dynamic programming works precisely in those scenarios that have the all-important *optimal substructure property*:

Property 5.8.1 (Optimal Substructure Property) *Dynamic programming works in those optimization settings, where a larger problem can be broken down into smaller subproblems of an identical nature. In other words, every optimal solution to the larger problem must also contain optimal solutions to the smaller subproblems.*

Here, the key point is that even though the number of solutions is extremely large, the optimal substructure property allows us to consider only a small subset of them. For example, the number of paths from the source to sink in a graph may be exponentially large, but one can easily compute all shortest paths containing at most 2 nodes from the source to all nodes. Because of the optimal substructure property, these paths can be extended to paths containing at most 3 nodes in linear time. This process can be repeated for an increasing number of nodes, until the number of nodes in the graph is reached. One generally implements dynamic programming via an iterative table-filling approach where smaller subproblems are solved first and their solutions are saved. Larger problems are then solved

as a function of the known solutions of the smaller problems using the optimal substructure property. In order to elucidate this point, we will use the example of optimizing the number of operations in chain matrix multiplication.

5.8.4.1 Application: Fast Matrix Multiplication

Consider the problem of multiplying the matrices A_1, A_2, A_3, A_4, and A_5 in that order. Because of the associative property of matrix multiplication, one can group the multiplications in a variety of ways without changing the result (as long as the sequential order of matrices is not changed). For example, one can group the multiplication as $[(A_1A_2)(A_3A_4)](A_5)$, or one can group the multiplication as $[(A_1)(A_2A_3)](A_4A_5)$. Consider the case where each A_i for odd i is a 1×1000 matrix, and each A_i for even i is a 1000×1 matrix. In such a case, the first grouping will require only about 3000 scalar multiplications to yield the final result of size 1×1000. All intermediate results will be compact scalars. On the other hand, the second grouping will create large intermediate matrices of size 1000×1000, the computation of which will require a million scalar multiplications. Clearly, the way in which the nesting is done is critical to the efficiency of matrix multiplication.

The decision problem in this case is to choose the top level grouping, since the subproblems are identical and can be solved in a similar way. For example, the top-level grouping in the first case is $[A_1A_2A_3A_4](A_5)$, and the top-level grouping in the second case above is $[A_1A_2A_3](A_4A_5)$. There are only four possible top-level groupings, and one needs to compute the number of operations in each case and choose the best among them. For each grouping, the smaller subproblems like $[A_1A_2A_3]$ and (A_4A_5) also need to be solved optimally. The complexity of multiplying the two intermediate matrices like $A_1A_2A_3$ and A_4A_5 of size $p \times q$ and $q \times r$, respectively, is pqr. This overhead is added to the complexity of the two subproblems to yield the complexity of that grouping.

Consider the matrices $A_1A_2 \ldots A_m$, where the matrix A_i is of size $n_i \times n_{i+1}$, and the optimal number of operations required for multiplying matrices i through j is $N[i, j]$. This leads to the following *dynamic programming recursion* for computing $N[1, m]$:

$$N[i, j] = \min_{k \in [i+1, j]} \{N[i, k-1] + N[k, j] + n_i n_k n_j\} \tag{5.36}$$

Note that the values on the right-hand side are computed earlier than the ones on the left using *iterative table filling*, where we compute all $N[i, j]$ in cases where $(j - i)$ is 1, 2, and so on in that order till $j - i$ is $(m - 1)$. There are at most $O(m^2)$ slots in the table to fill, and each slot computation needs the evaluation of the right-hand side of Equation 5.36. This evaluation requires a minimization over at most $(m-1)$ possibilities, each of which requires two table lookups of the evaluations of smaller subproblems. Therefore, each evaluation of Equation 5.36 requires $O(m)$ time, and the overall complexity is $O(m^3)$. One can summarize this algorithm as follows:

Initialize $N[i, i] = 0$ and $\text{Split}[i, i] = -1$ for all i;
for $\delta = 1$ to $m - 1$ **do**
  for $i = 1$ to $m - \delta$ **do**
    $N[i, i + \delta] = \min_{k \in [i+1, i+\delta]} \{N[i, k-1] + N[k, i + \delta] + n_i n_k n_{i+\delta}\}$;
    $\text{Split}[i, i + \delta] = \text{argmin}_{k \in [i+1, i+\delta]} \{N[i, k-1] + N[k, i + \delta] + n_i n_k n_{i+\delta}\}$;
  endfor;
endfor

One also needs to keep track of the optimal split position for each pair $[i, j]$ in a separate table $\text{Split}[i, j]$ in order to reconstruct the nesting. For example, one will first access $k = \text{Split}(1, m)$ in order to divide the matrix into two groups $A_1 \ldots A_{k-1}$ and $A_k \ldots A_m$.

Subsequently Split$[1, k-1]$ and Split$[k, m]$ will be accessed again to find the top-level nesting for the individual subproblems. This process will be repeated until we reach singleton matrices.

The word "dynamic programming" is used in settings beyond pure optimization. Many types of iterative table filling that achieve polynomial complexity by avoiding repeated operations are considered dynamic programming (even when no optimization occurs). For example, the backpropagation algorithm (cf. Chapter 11) uses the summation operation in the dynamic-programming recursion, but it is still considered dynamic programming. One can easily change the shortest-path algorithm between a source-sink pair to an algorithm for finding the number of paths between a source-sink pair (in a graph without cycles) with a small change to the form of the key table-filling step. Instead of computing the shortest path using each incident node i on source node s, one can compute the sum of the paths from each incident node i (on the source) to the sink. The key point is that an *additive* version of the substructure property holds, where the number of paths from source to sink is to equal to the sum of the number of paths from node i (incident on source) to sink. However, this is not an optimization problem. Therefore, the dynamic programming principle can also be viewed as a general computer programming paradigm that works in problem settings beyond optimization by exploiting any version of the substructure property — in general, the substructure property needs to be able to compute the statistics of superstructures from those of substructures via bottom-up table filling.

5.9 Summary

This chapter introduces a number of advanced methods for optimization, when simpler methods for gradient descent are not very effective. The simplest approach is to modify gradient descent methods, and incorporate several ideas from second-order methods into the descent process. The second approach is to directly use second-order methods such as the Newton technique. While the Newton technique can solve quadratic optimization problems in a single step, it can be used to solve non-quadratic problems with the use of local quadratic approximations. Several variations of the Newton method, such as the conjugate gradient method and the quasi-Newton method, can be used to make it computationally efficient. Finally, non-differentiable optimization problems present significant challenges in various machine learning settings. The simplest approach is to change the loss function to a differentiable surrogate. Other solutions include the use of the subgradient and the proximal gradient methods.

5.10 Further Reading

A discussion of momentum methods in gradient descent is provided in [106]. Nesterov's algorithm for gradient descent may be found in [97]. The delta-bar-delta method was proposed by [67]. The AdaGrad algorithm was proposed in [38]. The RMSProp algorithm is discussed in [61]. Another adaptive algorithm using stochastic gradient descent, which is *AdaDelta*, is discussed in [139]. This algorithms shares some similarities with second-order methods, and in particular to the method in [111]. The Adam algorithm, which is a further enhancement along this line of ideas, is discussed in [72]. The strategy of *Polyak averaging* is discussed in [105].

A description of several second-order gradient optimization methods (such as the Newton method) is provided in [19, 66, 83]. The implementation of the SVM approach with the